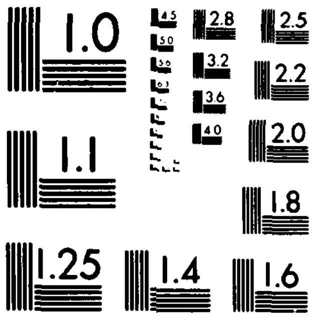
AD-A082 359   WEIZMANN INST OF SCIENCE REHOVOTH (ISRAEL) DEPT OF --ETC F/G 9/2
SYNTHESIZED STRUCTURED PROGRAMMING.(U)
JAN 80 Z MANNA   AFOSR-78-3483
UNCLASSIFIED   RADC -TR-79-326   NL

1 of 2
AD
A082 359


1.0
4.5
5.0
5.6
6.3
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19 REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 18 RADC-TR-79-326	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 SYNTHESIZED STRUCTURED PROGRAMMING.		5. TYPE OF REPORT & PERIOD COVERED 9 Final Technical Report. 1 Aug 75 — 30 Sep 78.
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) 10 Zohar Manna		8. CONTRACT OR GRANT NUMBER(s) 15 AFOSR-78-3483
9. PERFORMING ORGANIZATION NAME AND ADDRESS Weizmann Institute of Science Applied Mathematics Department Rehovot, Israel		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 16 55580835 17 Ø2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research (NM) Bldg 410 Bolling AFB Wash DC 20332	11	12. REPORT DATE Janu[redacted]980
		13. NUMBER OF PAGES 188 12 188
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Rome Air Development Center (ISIS) Griffiss AFB NY 13441		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Northrup Fowler, III (ISIS)

Prepared in cooperation with N. Dershowitz

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

program modification	program annotation
program debugging	
program instantiation	
program abstraction	
program synthesis	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Techniques of program modification are formulated, and an experimental software system implemented, whereby a given program that achieves one goal can be transformed into a new program to achieve a different goal. The essence of the approach is to find an analogy between the specifications of the given program and of the desired program, and then to transform the given program accordingly.

Program debugging is considered as a special case of modification: if a program computes wrong results, it must be modified to achieve the intended results

next page

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

110 763

JC

Item 20 (Cont'd)

cont. The abstraction of a set of concrete programs to obtain a program schema and the instantiation of abstract schemata to solve concrete problems are also viewed from the perspective of modification techniques.

Two tools are developed as aids in the above tasks: We describe transformation rules for synthesizing code from specifications in a top-down manner. They may be used when - in the course of modifying a program - the need arises to completely rewrite a program segment. For the purpose of determining what an incorrect program actually does - before attempting to debug it - we develop techniques of program annotation. These techniques are expressed as inference rules and derive invariant assertions from the program text.

These notes were prepared jointly by N. Dershowitz and Z. Manna. They are based on the papers by Dershowitz and Manna (1977, 1978) and Dershowitz's PH.D. Thesis (1978).

EVALUATION

This research effort forms part of a broad integrated project under RADC TPO No. 5 (C^3 System Availability), Thrust A (Software Cost Reduction), to attack the problem of spiraling Air Force software life-cycle costs. Complemented within the RADC project both technologically and temporally, this medium long range, high payoff effort focuses on automated software synthesis and modification. It develops theoretical techniques utilizing program annotation, abstraction, instantiation and transformation. These techniques, and the emerging theory that unifies them, form the basic substructure of future undertakings that will eventually result in automated production line aids for efficient software production and maintenance.

[illegible]

NORTHRUP FOWLER III
Project Engineer

CHAPTER I

INTRODUCTION

Programming begins with the specification of what the desired program should do; the programmer's job is to develop an executable program satisfying those specifications. The goal of automatic-programming research is to formalize the methods and strategies used by programmers so that they may be incorporated in an automatic, or interactive, programming environment.

While most automatic-programming research has focused on the creation of programs *ex nihilo*, very little of this work concentrates on applying past experience to new problems. Typically, a programmer directs more of his effort at the modification of programs that have already been written than at the development of original programs. The evolutionary cycle of a program includes debugging, changes to meet amended specifications, and extensions for expanded capabilities. Even when nominally engaged in the construction of a new program, the programmer is constantly recycling "used" programs and adapting basic principles that have already been incorporated into other programs. Ideas of general applicability are abstracted into subroutines or programming techniques and then applied to specific problems at hand.

In this research, we have attempted to emulate the evolutionary aspects of programming in the context of an automatic program-development system. We have formulated techniques of *program modification*, whereby a given program that achieves one goal can be transformed into a new program to achieve a different goal. The essence of the approach is to find an *analogy* between two sets of specifications, those of a program that has already been constructed and those of the program that we desire to construct. This analogy is then used as the basis for transforming the existing program to meet the new specifications. *Program debugging* is considered as a special case of modification: if a program computes wrong results, it must be modified to achieve the intended results.

Program modification is not the only manner in which a programmer utilizes previously acquired knowledge. The human programmer improves with experience by assimilating various programming methods that he encounters, and judiciously applying the learned ideas to new problems. After coming up with several modifications of his first "wheel", he is likely to formulate for himself (and perhaps for others) an abstract notion of the underlying principle and reuse it in new, but related, applications. *Program schemata* are a convenient form for remembering such programming knowledge. A schema may embody basic programming techniques and strategies (e.g. the generate-and-test paradigm or the binary-search technique) and contains abstract predicate, function, and constant symbols, in terms of which its specification is stated.

The *abstraction* of a set of concrete programs to obtain a program schema and the *instantiation* of abstract schemata to solve concrete problems may be viewed from the perspective of modification techniques. This perspective provides a methodology for applying old knowledge to new problems. Beginning with a set of programs sharing some basic strategy and their correctness proofs, a program schema that represents the embedded technique is sought. Preconditions for the schema's applicability are also derived from the correctness proofs. The schema's abstract specification may then be compared with a given concrete specification and an instantiation found that, when applied to the schema, yields a concrete program. If the instantiation satisfies the preconditions, then the correctness of the new program is guaranteed.

Extending a program to satisfy additional specifications is another form of program modification. Techniques are required to construct code that extends the incomplete program to achieve the remaining specifications, while ensuring that the original specifications continue to be satisfied. Modification based on analogy and extension can be combined to solve a given problem. The analogy between a new problem and a given program may only indicate how to achieve part of the specified goal; the transformed program is then extended to achieve the remainder.

Sometimes, in the course of modifying a program or instantiating a schema, it may turn out that a program segment, e.g. a loop initialization, must be constructed from scratch. Top-down *synthesis techniques* are useful for this purpose. Beginning with the specifications of the desired segment, the goal is to develop the program step by step until executable code is obtained. Each step consists of rewriting a segment of the program in increased detail. Since every step is transparent enough to ensure correctness, each partial program in the series is equivalent to its predecessor. In particular, the final program is guaranteed to satisfy the initial specifications.

A prerequisite for debugging an incorrect program is knowledge about what the program actually does, as opposed to what it was intended to do. Moreover, various facts about a program are frequently needed for the purposes of modification, though they were not supplied by the programmer. For these purposes, we devote attention to the development of *annotation techniques* for documenting a program with *assertions*. Assertions are a useful means of documenting facts about the internal workings of a program; they relate to specific points in the program and assert that some relation holds for the current values of the program variables whenever control passes through that point. Given a program along with its input-output specification, the task is to annotate the program incrementally with assertions that explain the actual workings of the program regardless of whether the program is correct. These annotations can be used as aids in the debugging of an incorrect program. They can also be used for verifying the correctness of programs or for analyzing program efficiency. Our annotation techniques are formulated as inference rules.

The techniques of program manipulation that we have investigated are for the most part amenable to automation, and we have implemented them in an experimental system, written in QLISP. Our implementation consists of three parts: *modifier*, *annotator*, and *synthesizer*. The implementation was meant to serve as a proving ground for ideas; many of the examples presented in this report have run successfully. The modifier has, for example, modified an integer square-root program to compute quotients and has debugged an incorrect real-division program. Our annotator can generate the necessary invariants for these programs, and for more complex programs, e.g. selection sort. The synthesizer has successfully constructed several complete programs, such as one for finding the minimal element of an array, or for finding its value.

The next chapter presents a general overview of the various aspects of program modification; their individual roles and their close interaction are illustrated in an account of the evolution of an example program. The remainder of this report is composed of chapters on techniques for

- modification and debugging,
- abstraction and instantiation,
- synthesis, and
- annotation.

Each of these chapters is largely self-contained, though a common set of examples is threaded through them. Bibliographic remarks are included in the individual sections.

CHAPTER II

GENERAL OVERVIEW

In this overview we shall trace the life-cycle of a single example program, in an attempt to impart the overall flavor of our approach to program modification, and to illustrate how the various aspects are interrelated. More formal treatments of our techniques may be found in the individual chapters. This example is outlined in Figure 1; it owes its motivation to Wensley [1959] and Dijkstra [1976].

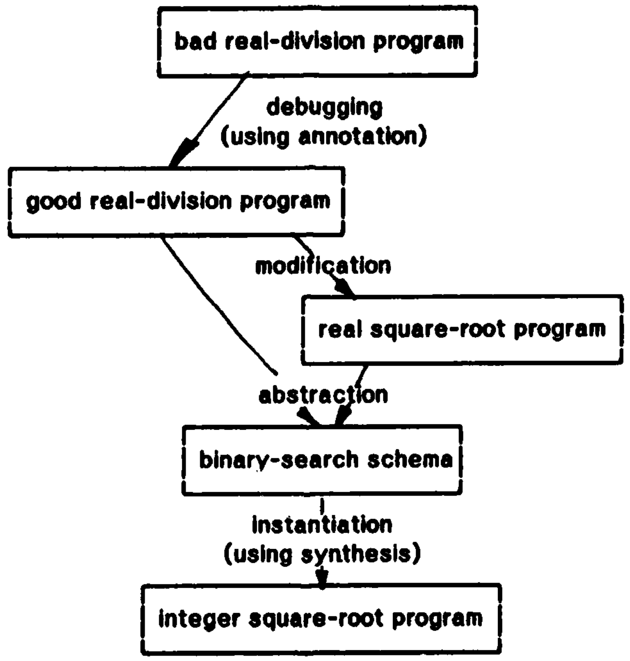


Figure 1. *Evolution of a division program*

We begin with an imperfect program to compute the quotient of two real numbers. We then *debug* the program, after determining enough about what the program actually does. Once the division program is corrected, it is *modified* to compute the square-root of a real number. Underlying both the division and square-root program is the binary-search technique; by *abstracting* these two programs, a binary-search schema is obtained. This schema is then *instantiated* to obtain a third program, one to compute the square-root of an integer. Part of that program is synthesized from scratch.

1. *The Problem*

Consider the problem of computing the quotient z of two nonnegative real numbers c and d within a specified (positive) tolerance e. These specifications are conveniently expressed in a high-level *assertion language* in terms of an *output specification* and an *input specification*. The output specification states the desired relationship among the program variables upon termination. In our case, the output specification

$|c/d-z|<e$

indicates that the (absolute value of the) difference between the exact value of c/d and the result z should be less than e. The input specification defines the set of inputs on which the program is intended to operate. Assuming that we only wish to solve this problem for the case where the numerator c is smaller than the denominator d, the appropriate input specification for the program is

$0 \le c < d \wedge e > 0$.

We can express our goal in the form of the following skeleton program:

```
P₁: begin comment real-division program
    assert 0≤c<d, e>0
    achieve |c/d-z|<e varying z
    end .
```

The **achieve** statement,

achieve $|c/d-z|<e$ **varying** z ,

specifies the relation between the variables z, c, d, and e that we wish to attain at the end of program execution. The clause

varying z

indicates that only the variable z may be set by the program; the variables c, d, and e contain input values that may not be modified. The **assert** statement,

assert $0 \le c < d,\ e > 0$,

attached to the beginning of the program, specifies what relation between the input variables may be assumed to always hold at the beginning of program execution.

An **achieve** statement may be considered as a "very high-level" programming construct that "somehow" achieves the specified relation at that point in the program. It is not directly executable; the task of the programmer — be he human or machine — is to systematically transform the **achieve** statement into an executable program by replacing it with more concrete code. If the replacement itself contains **achieve** statements, then the process iterates, step by step, until a machine-executable program is obtained that contains only primitive statements and operators. This final program will be of the form

```
P1: begin comment real-division program
    assert 0≤c<d, e>0
    purpose |c/d-z|<e
        code
    suggest |c/d-z|<e
    end .
```

The **purpose** statement,

purpose $|c/d-z|<e$,

is a comment describing what the intent of the code following it is. The statement

suggest $|c/d-z|<e$

contains the programmer's contention that the preceding code actually achieves the desired relation, i.e. the relation $|c/d-z|<e$ holds for the value of z when control reaches the end of the program.

When an assertion, such as $|c/d-z|<e$, has been *proved* to hold each time control passes through some point, then it is said to be an *invariant assertion* at that point. As long as it has not been proved to hold, it is called a *candidate*. In particular, an *output candidate*, associated with the point of termination, is a local invariant at that point, if the final values of the variables satisfy the asserted relation when the program terminates. The assertion is termed an *output invariant* once this has been proved to be the case. A program, then, may be considered correct if there exist output invariants that imply the output specification.

For the problem at hand, we must assume that no general real-division operator / is available, though division by an integer is permissible. Otherwise, the problem could be solved with a trivial assignment statement

$z := c/d$.

The reader may also note that, were it not for the restriction that only the variable z may be set by the program, the problem could be solved, for example, by setting both z and c to 0 . This would satisfy the specification $|c/d-z|<e$, but is not the intended solution.

Now let us assume that a programmer went ahead and constructed the following program:

```
P₁: begin comment suggested division program
    B₁: assert 0≤c<d, e>0
    purpose |c/d-z|<e
         purpose z≤c/d, c/d<z+y, y≤e
         (z,y) := (0,1)
         loop L₁: suggest z≤c/d, c/d<z+y
              until y≤e
              if d·(z+y)≤c then z := z+y fi
              y := y/2
              repeat
         suggest z≤c/d, c/d<z+y, y≤e
    E₁: suggest |c/d-z|<e
    end .
```

The comment

purpose $z \le c/d,\ c/d < z+y,\ y \le e$

indicates that the programmer's intention is to achieve the desired relation $|c/d-z|<e$ by achieving the three subgoals $z \le c/d$, $c/d<z+y$, and $y \le e$. Achieving these relations is sufficient for $|c/d-z|<e$ to hold. To achieve them, the programmer constructed an iterative loop intended to keep the first two relations invariantly true while making progress towards the third. The intended loop invariants are given in the statement

suggest $z \le c/d,\ c/d<z+y$

at the label L_1 ; they are first initialized by the multiple assignment

(z,y) := $(0,1)$,

since both $0 \le c/d$ and $c/d<0+1$ are implied by the assumption that $0 \le c<d$. The two loop-body statements

```
if d·(z+y)≤c then z := z+y fi
y := y/2
```

are then repeated until the test

until $y \le e$

becomes true, at which point the loop is left.*

For the candidates

suggest $z \leq c/d,\ c/d < z+y$

to be loop invariants, they must hold when the loop is first entered and must remain true each subsequent time control returns to the beginning of the loop. Though we have seen that they do hold initially, it has not yet been verified that they remain true. Consequently, it is also not known if the output candidate $|c/d-z|<e$ is invariant. In fact, by running the program with $c=1$, $d=3$, and $e=1/3$, for instance, the programmer may discover that the result $z=0$ does not satisfy $|c/d-z|<e$. Since these values for the variables satisfy the input specification, but do not satisfy the output specification, the program is incorrect. The bug presumably occurred when the programmer "inadvertently" interchanged the two statements within the loop.

2. *Annotation*

We know what this program was intended to do. However, before we can debug it, we must know more about what it actually does. This will be accomplished by examining the code, trying to extract as many relations between the variables as we can, and annotating the program with the discovered relations.

Our techniques for program annotation are discussed in more detail in the chapter on annotation. There they are expressed as inference rules: the antecedents of each rule are usually annotated program segments and the consequent is either an invariant or a candidate. These rules have been implemented; the automatic annotation of a similar program is shown in the appendix on implementation.

As a first step, we note that the input variables c, d, and e are not changed by the program. Therefore the input assertion

assert $0 \leq c < d,\ e > 0$

holds throughout execution of the program. Such an assertion is termed a *global invariant* of the program; we write

assert $0 \leq c < d,\ e > 0$ **in** P_1 .

The loop-until-repeat construct we use is based on the suggestion of J. Ole-Dahl in Knuth [1974]; *achieve statements were used by Sussman* [1975].

We now try to determine the range of the two program variables y and z. The assignments to y in the program P_1 are

$$y := 1 \qquad y := y/2 .$$

The variable y is initialized to 1 before the loop and is repeatedly divided by 2 within the loop. It follows that $y=1/2^n$, where n is some nonnegative integer indicating the number of times that y has been halved.

In dealing with sets, we find the following notation convenient: Let $f(s_1, s_2, \ldots, s_m)$ be any expression containing occurrences of m distinct subexpressions $s_1, s_2, \ldots, s_m$. The set of elements

$$\{ f(s_1, s_2, \ldots, s_m) \colon s_1 \in S_1, s_2 \in S_2, \ldots, s_m \in S_m \}$$

is denoted by

$$f(S_1, S_2, \ldots, S_m) .$$

Using this notation, we say that y belongs to the set $1/2^{\mathbb{N}}$, where $\mathbb{N}$ is the set of nonnegative integers. Since this relation holds throughout the program P_1 from the point when the assignment $y{:=}1$ is first executed, we may assert the global invariant

$$\textbf{assert } y \in 1/2^{\mathbb{N}} \textbf{ in } P_1 .$$

From this invariant one can derive both an upper and lower bound on y. At one extreme $y=1/2^0=1$, and at the other extreme — as the exponent increases — the value of y approaches 0. Thus, we

$$\textbf{assert } 0<y\le 1 \textbf{ in } P_1 .$$

The program contains two assignments to the variable z,

$$z{:=}0 \qquad z{:=}z+y .$$

Since we have already determined that y is always of the form $1/2^n$, it follows that z must be a sum of some finite number (possibly zero) of elements of that form. This does not tell too much about z; it does, though, give the lower bound

$$\textbf{assert } z\ge 0 \textbf{ in } P_1 ,$$

since y is always positive.

The loop terminates when the exit test $y \le e$ becomes true. Thus, whenever control reaches the label E_1, the relation $y \le e$ must hold. This is expressed by the *local invariant*

E_1: **assert** $y \le e$.

Similarly, if the exit test is not taken and the loop body is executed, then the exit test must have been false, i.e. $y > e$.

Neither branch of the conditional statement affects y, and therefore the relation $y > e$ holds after the conditional statement as well. At that point y is divided by 2 . If before the division we had $y > e$, then at the end of the loop body we have $2 \cdot y > e$. So, whenever the loop body is executed control returns to the head of the loop with the relation $2 \cdot y > e$ holding. Since that relation does not necessarily hold when the loop is first entered with $y = 1$, it is not a loop invariant. Nevertheless, the disjunction of the relations $y = 1$ and $2 \cdot y > e$ is a loop invariant, since one relation holds when the loop is first entered and the other holds every time the loop is repeated, i.e. we have

L_1: **assert** $y = 1 \vee 2 \cdot y > e$.

Consider the conditional statement

if $d \cdot (z+y) \le c$ **then** $z := z+y$ **fi** .

It is an abbreviation of the statement

if $d \cdot (z+y) \le c$ **then** $z := z+y$ **else** **fi**

which has an empty **else**-branch. The **then**-path of the conditional statement is taken when $d \cdot (z+y) \le c$; therefore, after resetting z to $z+y$ we have $d \cdot z \le c$. Since the programmer introduced the conditional statement to achieve some specific relation in different cases, it is plausible that the relation $d \cdot z \le c$ — achieved by the **then**-path of the conditional — is the intended relation and holds for the **else**-path as well. This suggests the candidate

L_1: **suggest** $d \cdot z \le c$.

Indeed, since $d \cdot z \le c$ is true initially, when $z=0$ and $c \ge 0$, and is unaffected when the conditional test is false (since the value of z is not changed), it invariantly holds when control reaches the head of the loop. We have derived the loop invariant:

L_1: **assert** $d \cdot z \le c$.

The **then**-path is not taken when $c < d \cdot (z+y)$. In that case y is divided in half and z

is left unchanged, yielding $c<d\cdot(z+2\cdot y)$ at the end of the current iteration. It turns out that the **then**-path preserves this relation and that it also holds upon initialization. Thus we have the additional invariant:

L_1: **assert** $c<d\cdot(z+2\cdot y)$.

The loop invariants $d\cdot z\leq c$ and $c<d\cdot(z+2\cdot y)$ remain true when the loop exited is taken; along with the exit test $y\leq e$, they imply that upon termination of the program the output invariant

E_1: **assert** $|c/d-z|<2\cdot e$

holds. Note that the desired relation $|c/d-z|<e$ is *not* implied.

The annotated program — with invariants that correctly express what the program does — is:

```
assert $0\leq c<d,\ e>0,\ y\in 1/2^N,\ z\geq 0$ in
$P_1$: begin comment annotated bad division program
    $B_1$: assert $0\leq c<d,\ e>0$
    $(z,y)$ := $(0,1)$
    loop $L_1$: assert $d\cdot z\leq c,\ c<d\cdot(z+2\cdot y),\ y=1 \lor 2\cdot y>e$
        suggest $d\cdot z\leq c,\ c<d\cdot(z+y)$
        until $y\leq e$
        if $d\cdot(z+y)\leq c$ then $z$ := $z+y$ fi
        $y$ := $y/2$
        repeat
    $E_1$: assert $|c/d-z|<2\cdot e$
    suggest $|c/d-z|<e$
    end .
```

We have omitted the **purpose** statements to avoid clutter.

3. *Debugging*

Now that we know something about what the program does, we can try to debug it. Our task is to find a correction that transforms the actual output invariant

assert $|c/d-z|<2 \cdot e$

into the desired output candidate

suggest $|c/d-z|<e$.

We shall then apply that *transformation* to the program in an attempt to derive a correct program.

Accordingly, we would like to modify the program in such a manner as to transform the insufficient $|c/d-z|<2 \cdot e$ into the desired $|c/d-z|<e$; we write

$|c/d-z|<2 \cdot e \Rightarrow |c/d-z|<e$.

The obvious difference between the two expressions, is that where the first has $2 \cdot e$, the second has just e . So, to transform $|c/d-z|<2 \cdot e$ into $|c/d-z|<e$, we need only transform

$2 \cdot e \Rightarrow e$,

leaving the other symbols unchanged. This may be accomplished by replacing e with $e/2$, i.e. by applying the transformation $e \Rightarrow e/2$. In this manner, we get

$|c/d-z|<2 \cdot e \quad \Rightarrow \quad |c/d-z|<2 \cdot e/2 \equiv |c/d-z|<e$.

We see that the transformation $e \Rightarrow e/2$, applied to the output invariant $|c/d-z|<2 \cdot e$, yields the desired output specification $|c/d-z|<e$. That same transformation is now applied to the whole annotated program (excluding the programmer's suggestions). The symbol e appears once in the program text: the exit clause

until $y \leq e$

accordingly becomes

until $y \leq e/2$.

The symbol also appears four times in the invariants; for example, the input assertion $e>0$ transforms into $e/2>0$ which is equivalent to $e>0$.

The transformed program is

```
assert $0 \le c < d$, $e > 0$, $y \in 1/2^N$, $z \ge 0$ in
$P_2$: begin comment corrected division program
    $B_2$: assert $0 \le c < d$, $e > 0$
    $(z, y)$ := $(0, 1)$
    loop $L_2$: assert $d \cdot z \le c$, $c < d \cdot (z + 2 \cdot y)$, $y = 1 \vee 4 \cdot y > e$
        suggest $d \cdot z \le c$, $c < d \cdot (z + y)$
        until $y \le e/2$
        if $d \cdot (z + y) \le c$ then $z$ := $z + y$ fi
        $y$ := $y/2$
        repeat
    $E_2$: assert $|c/d - z| < e$
    suggest $|c/d - z| < e$
    end .
```

In an appendix, it is proved a transformation such as $e \Rightarrow e/2$ preserves the relation between the program text and invariants, i.e. the transformed assertions are invariants of the transformed program.

In this manner, we have modified the program to achieve the intended result $|c/d - z| < e$. But note that the loop invariant still differs from that suggested by the programmer. The difference between the two is that the programmer intended for $c < d \cdot (z + y)$ to be true, while in fact $c < d \cdot (z + 2 \cdot y)$ holds. This can be remedied by applying the transformation

$y \Rightarrow y/2$.

The variable y appears five times in the program code: The exit clause becomes

until $y/2 \le e/2$,

or equivalently

until $y \le e$.

The conditional statement becomes

if $d \cdot (z + y/2) \le c$ **then** z := $z + y/2$ **fi** .

The assignment statement

y := 1

transforms into

$y/2$:= 1 ,

which, however, is not a legal assignment, since an expression appears on the left-hand side. The intent of this illegal statement is to

achieve $y/2=1$ **varying** y .

By multiplying the two sides of the equality by 2 , it is seen to be equivalent to

achieve $y=2$ **varying** y ,

which may be accomplished by the assignment

$y := 2$.

Similarly, the original assignment

$y := y/2$

gives rise to the goal

achieve $y/2 = (y'/2)/2$ **varying** y ,

where y' represents the prior value of the variable y . Again, by multiplying both sides by 2 , we derive the assignment

$y := y/2$.

Thus, we have obtained the program:

```
assert 0≤c<d, e>0, y∈1/2^N, z≥0 in
P₂': begin comment transformed division program
    B₂: assert 0≤c<d, e>0
    (z,y) := (0,2)
    loop L₂: assert d·z≤c, c<d·(z+y), y=2∨2·y>e
        suggest d·z≤c, c<d·(z+y)
        until y≤e
        if d·(z+y/2)≤c then z := z+y/2 fi
        y := y/2
        repeat
    E₂: assert |c/d-z|<e
    suggest |c/d-z|<e
    end .
```

Since the expression $y/2$ appears thrice in the loop body, this program may be slightly improved by evaluating the subexpression $y/2$ before the conditional statement. We obtain:

```
assert $0 \le c < d$, $e > 0$, $y \in 1/2^N$, $z \ge 0$ in
$P_2''$: begin comment good division program
    $B_2$: assert $0 \le c < d$, $e > 0$
    $(z, y) := (0, 2)$
    loop $L_2$: assert $d \cdot z \le c$, $c < d \cdot (z+y)$, $y = 2 \vee 2 \cdot y > e$
        until $y \le e$
        $y := y/2$
        if $d \cdot (z+y) \le c$ then $z := z+y$ fi
        repeat
    $E_2$: assert $|c/d - z| < e$
    end .
```

Note that this program is almost the same as the original bad program. It differs in two ways: the two loop-body assignments are interchanged (this presumably was the error), and y is initialized to 2 rather than 1 (either initialization works).

4. *Modification*

Consider the following specifications:

```
$P_3$: begin comment square-root program
    assert $a \ge 1$, $e > 0$
    achieve $|\sqrt{a} - z| < e$ varying $z$
    end .
```

We would like to use the corrected real-division program as a basis for the construction of the specified program for computing square-roots. We assume that the $\sqrt{\ }$ operator is not primitive.

To this end, we first compare the specifications of the two programs. The output specification of the division program is

assert $|c/d - z| < e$

while the output specification of the desired program is

achieve $|\sqrt{a} - z| < e$ **varying** z .

The obvious analogy between the two is

$$c/d \Leftrightarrow \sqrt{a} \ ,$$

i.e. where one has c/d, the other has $\sqrt{a}$. Thus, to obtain a square-root program from the division program, we need to transform c/d into $\sqrt{a}$. One way to do this would be via the transformations

$$(d \Rightarrow 1, c \Rightarrow \sqrt{a}) \ ,$$

which take c/d into $\sqrt{a}/1=\sqrt{a}$. Here d is transformed into the identity element of the division operator, leaving c to become $\sqrt{a}$. Alternatively, we could apply the transformation

$$(u/v \Rightarrow \sqrt{u}, c \Rightarrow a)$$

where by $u/v \Rightarrow \sqrt{u}$ we mean that every occurrence of the division operator is replaced by the square-root operator applied to what was the numerator.

We apply the *first* set of transformations to the division program P_2, annotated with only those invariants essential for proving correctness. Replacing all occurrences of d with 1 and all occurrences of c with $\sqrt{a}$ and simplifying, yields

```
assert  $0\le\sqrt{a}<1,\ e>0$  in
$P_3$: begin  comment  square-root program
    $B_3$: assert  $0\le\sqrt{a}<1,\ e>0$
    $(z,y)$ := $(0,2)$
    loop $L_3$: assert  $z\le\sqrt{a},\ \sqrt{a}<z+y$
        until  $y\le e$
        $y$ := $y/2$
        if  $z+y\le\sqrt{a}$  then  $z$ := $z+y$  fi
        repeat
    $E_3$: assert  $|\sqrt{a}-z|<e$
    end .
```

The transformed program is guaranteed to satisfy the output specification $|\sqrt{a}-z|<e$; unfortunately, it is inexecutable inasmuch as it contains the nonprimitive function $\sqrt{\ }$ in the conditional test.

It is assumed that knowledge about the subject domain is available. For example, we need the following fact about the square-root function:

fact $u\le\sqrt{v} \equiv u^2\le v$ **when** $u\ge0$, where u and v are universally quantified, i.e.

$$(\forall u,v)\ (u \geq 0 \supset u \leq \sqrt{v} \equiv u^2 \leq v)\ .$$

This fact allows us to replace the test $z+y \leq \sqrt{a}$ with the equivalent $(z+y)^2 \leq a$. That $z+y$ is indeed nonnegative, as required for the two tests to be equivalent, follows from the loop invariant

```
$L_5$: assert $\sqrt{a}<z+y$
```

and the

```
fact $0 \leq \sqrt{u}$ .
```

There remains an additional problem: a transformed program is only guaranteed to satisfy the output specification for those inputs that satisfy the transformed input specification. Unfortunately, the transformed input specification of our program,

```
assert $0 \leq \sqrt{a} < 1$, $e>0$ ,
```

is contrary to the given input specification $a \geq 1$. To solve this, we can replace the code preceding the loop with the goal

```
assert $a \geq 1$, $e>0$
achieve $z \leq \sqrt{a}$, $\sqrt{a}<z+y$ varying $z,y$ .
```

to initialize the loop invariants $z \leq \sqrt{a}$ and $\sqrt{a}<z+y$ prior to entering the loop. Achieving $z^2 \leq a$ is equivalent to achieving $z \leq \sqrt{a}$. And since it is given that $1 \leq a$, we need only achieve $z^2=z=1$. To achieve the second conjunct $\sqrt{a}<z+y$, given the

```
fact $\sqrt{u} \leq u$ when $u \geq 1$ ,
```

we need only achieve $z+y=1+y=a$. Thus we have reduced the goal to

```
achieve $z=1$, $1+y=a$ varying $z,y$ ,
```

giving rise to the assignment

```
$(z,y)$ := $(1,a-1)$ .
```

The square-root program that we have obtained is:

```
assert $a \geq 1,\ e>0$ in
$P_3$: begin comment square-root program
    $B_3$: assert $a \geq 1,\ e>0$
    $(z,y)$ := $(1,a-1)$
    loop $L_3$: assert $z \leq \sqrt{a},\ \sqrt{a}<z+y$
        until $y \leq e$
        $y$ := $y/2$
        if $(z+y)^2 \leq a$ then $z$ := $z+y$ fi
        repeat
    $E_3$: assert $|\sqrt{a}-z|<e$
    end .
```

Note that the global invariant $0 \leq \sqrt{a} < 1$ no longer holds.

The alternative set of transformations for transforming the division program into a square-root program was

$$(u/v \Rightarrow \sqrt{u}, c \Rightarrow a)\ .$$

Transformations that involve specific functions such as u/v, are not, however, guaranteed to yield a correct program, since the program may be based on some property that holds for u/v but not for $\sqrt{u}$. These transformations are heuristic in nature; they only suggest a possibly incomplete analogy between the two programs. Indeed, when applied to the division program, the transformations yield

```
suggest $0 \leq a < d,\ e>0$ in
$P_3'$: begin comment transformed division program
    $B_3'$: suggest $0 \leq a < d,\ e>0$
    $(z,y)$ := $(0,2)$
    loop $L_3'$: suggest $d \cdot z \leq a,\ a < d \cdot (z+y)$
        until $y \leq e$
        $y$ := $\sqrt{y}$
        if $d \cdot (z+y) \leq a$ then $z$ := $z+y$ fi
        repeat
    $E_3'$: suggest $|\sqrt{a}-z|<e$
    end ,
```

which clearly does not compute $\sqrt{a}$. Since this set of transformations is not correctness-preserving, the asserted invariants have been replaced by suggested candidates.

What must be done is to review the derivation of the program, expressed in the **purpose** statements and see where the analogy breaks down. The purpose of the division program is $|c/d-z|<e$ which transforms into $|\sqrt{a}-z|<e$ as desired. The programmer achieved $|c/d-z|<e$ by breaking it into the conjunction of three subgoals, given in the statement

purpose $c/d \geq z,\ c/d<z+y,\ y \leq e$

that appeared in the original program. The last conjunct became the exit test, and the other two became loop invariants. These subgoals transform into

purpose $\sqrt{a} \geq z,\ \sqrt{a}<z+y,\ y \leq e$,

which indeed imply the transformed goal $|\sqrt{a}-z|<e$.

The purpose of the loop body of the division program (though it was left out of P_1) was

purpose $c/d \geq z,\ c/d<z+y,\ 0<y<y_{L_2}$,

where y_{L_2} represents the value of the variable y when last at the head of the loop, at label L_2. In other words, the loop body reachieves the invariants while making progress towards the exit test by decreasing y (to guarantee termination, that decrease cannot be arbitrarily small). The loop-body subgoal of the transformed program, then, is

purpose $\sqrt{a} \geq z,\ \sqrt{a}<z+y,\ 0<y<y_{L_2}$.

The division program first decreases y and then introduces a conditional with the

purpose $c/d \geq z,\ c/d<z+y$.

It is here that the analogy breaks down. The loop body of the division program achieves this purpose in two cases, by testing if $d \cdot (z+y) \leq c$ or not. For example, if $d \cdot (z+y) \leq c$ does not hold, then $c/d<z+y$, as desired. On the other hand, the fact that $d \cdot (z+y) \leq a$ does not hold in the square-root program tells nothing about $\sqrt{a}<z+y$. We look, therefore, for a transformation that will allow the implication

$$d \cdot (z+y)>a \supset \sqrt{a}<z+y$$

to hold. As for the previous alternative, since $z+y$ is nonnegative, the right hand side of the implication is equivalent to $a<(z+y)^2$. Matching the left-hand side of the implication with this inequality, tells us that the implication would hold if we could transform $d \cdot (z+y) \Rightarrow (z+y)^2$. Thus, where the division program has the function $u \cdot v$, the square-root program requires v^2. We complete the analogy by adding the transformation $u \cdot v \Rightarrow v^2$, to obtain

$$(u/v \Rightarrow \sqrt{u}, c \Rightarrow a, u \cdot v \Rightarrow v^2) .$$

Finally, as with the first set of transformations, the initialization subgoal does not hold, and must be replaced by the assignment

$$(z,y) := (1,a-1) .$$

The same program

```
assert $a \geq 1,\ e > 0$ in
$P_3$: begin comment square-root program
    $B_3$: assert $a \geq 1,\ e > 0$
    $(z,y) := (1,a-1)$
    loop $L_3$: assert $z \leq \sqrt{a},\ \sqrt{a} < z+y$
          until $y \leq e$
          $y := y/2$
          if $(z+y)^2 \leq a$ then $z := z+y$ fi
          repeat
    $E_3$: assert $|\sqrt{a}-z| < e$
    end
```

is obtained.

5. *Abstraction*

We now have two programs, P_2 for finding the quotient and P_3 for finding the square-root. Both programs utilize the binary-search technique. We would like to extract an abstract version of the two programs that captures the essence of the technique, but that is not specific to either problem. The resultant abstract program schema can then be used as a model of binary search for the solution of future problems.

Consider the second analogy that we found between P_2 and P_3:

$$(u/v \Leftrightarrow \sqrt{u}, c \Leftrightarrow a, u \cdot v \Leftrightarrow v^2) .$$

Both u/v and $\sqrt{u}$ are functions; they may be generalized to some abstract function $f(u,v)$. Similarly the generalization of $u \cdot v$ and v^2 is $g(u,v)$. Since both c and a are variables, we can leave them as is. This gives us the following set of transformations to generalize the division program:

$$(u/v \Rightarrow f(u,v), u \cdot v \Rightarrow g(u,v)) .$$

Applying these transformations to the specifications

achieve $|c/d-z|<\epsilon$ **varying** z

of the division program yields

achieve $|f(c,d)-z|<\epsilon$ **varying** z .

This, then, shall be the abstract output specification of the schema. The division program was

```
P₂: begin comment good division program
    B₂: assert  0≤c<d,  ε>0
    (z,y) := (0,2)
    loop L₂: assert  d·z≤c,  c<d·(z+y)
         until  y≤ε
         y := y/2
         if  d·(z+y)≤c  then  z := z+y  fi
         repeat
    E₂: assert  |c/d-z|<ε
    end .
```

Substituting the abstract predicates f and g into their respective positions in the annotated program, we derive the schema:

```
P₄: begin comment transformed division program
    B₂: assert  0≤c<d,  ε>0
    (z,y) := (0,2)
    loop L₂: assert  g(d,z)≤c,  c<g(d,z+y)
         until  y≤ε
         y := y/2
         if  g(d,z+y)≤c  then  z := z+y  fi
         repeat
    E₂: assert  |f(c,d)-z|<ε
    end .
```

This schema is not necessarily correct for all instantiations of f and g, as the original program relied upon facts specific to / and Indeed, there is nothing in this abstract program to relate the function f that appears in the output specification with the function g that appears in the loop invariant. In general, transformations (abstractions) of specific function or predicates are not correctness-preserving. We would like then to determine under what conditions this abstract schema does achieve its specifications.

As in the modification step, the initialization assignment does not necessarily achieve the desired loop invariants. We therefore replace the initialization assignment with the subgoal

achieve $g(d,z) \le c,\ c < g(d,z+y)$ **varying** z, y ,

leaveing unspecified how to initialize the two loop invariants.

For the loop-body path to be correct, the truth of the invariant must imply that the invariant will hold the next time around; this can easily be shown to be the case for any function g. For the loop-exit path to be correct, we must have that the loop invariants $g(d,z) \le c$ and $c < g(d,z+y)$ plus the exit test $y \le e$ imply that the output invariant $|f(c,d)-z| < e$ holds. For this to be the case, it suffices to establish the condition

assert $g(w,u) \le v \equiv u \le f(v,w)$.

This assertion holds if g is the inverse of a monotonic function f, i.e. $f(g(w,u),w)=u$ and $(u \le v) \equiv (f(u,w) \le f(v,w))$, as, for example, • is the inverse of / and u^2 is the inverse of $\sqrt{u}$.

In this manner, we have derived a general program schema for a binary search for the value of $f(c,d)$ within a tolerance e :

```
P4: begin comment binary-search schema
    B4: assert $g(w,u) \le v \equiv u \le f(v,w)$
    achieve $g(z,d) \le c,\ c < g(z+y,d)$ varying $z, y$
    loop L4: assert $g(d,z) \le c,\ c < g(d,z+y)$
        until $y \le e$
        $y := y/2$
        if $g(d,z+y) \le c$ then $z := z+y$ fi
        repeat
    E4: assert $|f(c,d)-z| < e$
    end .
```

Its output specification is

assert $|f(c,d)-z|\le\epsilon$,

and its precondition for guaranteed correctness is

assert $g(w,u)\le v \equiv u\le f(v,w)$.

Clearly, the function g which appears in the schema must be primitive; otherwise, it must be replaced by something equivalent for the schema to yield an executable program. The unachieved subgoal

achieve $g(z,d)\le c,\ c<g(z+y,d)$ **varying** z,y

must also be reduced to primitives.

6. *Instantiation*

We illustrate how the binary-search schema just derived may be applied to the computation of integer square-roots. Our goal is to construct a program that finds the integer square-root z of a nonnegative integer a :

```
P_8: begin comment integer square-root program
      B_8: assert a∈N
      achieve z=⌊√a⌋ varying z
      end ,
```

where the function $\lfloor u\rfloor$ yields the greatest integer less than or equal to u .

We cannot directly match this goal with the output specification of the schema

assert $|f(c,d)-z|<\epsilon$ **varying** z ,

or with any of the other invariants known to hold upon termination of the schema. However, if we expand the goal $z=\lfloor\sqrt{a}\rfloor$, using the definition of $\lfloor u\rfloor$,

fact $v=\lfloor u\rfloor \equiv v\le u\wedge u<v+1\wedge v\in\mathbb{Z}$

(where $\mathbb{Z}$ is the set of all integers), we get the equivalent goal

achieve $z\le\sqrt{a},\ \sqrt{a}<z+1,\ z\in\mathbb{Z}$ **varying** z ,

i.e. z should be the largest integer not greater than $\sqrt{a}$. Since we know that the schema achieves the two output invariants

assert $z\le f(c,d),\ f(c,d)<z+\epsilon$,

we compare these invariants with the above goal. This suggests the transformation

$$(f(u,v) \Rightarrow \sqrt{u}, c \Rightarrow a, e \Rightarrow 1)$$

to achieve the two conjuncts $z \leq \sqrt{a}$ and $\sqrt{a} < z+1$; in addition, we will have to extend the program to ensure that the final value of z is a nonnegative integer. For this purpose, we append the subgoal

achieve $z \in Z$ **protecting** $z \leq \sqrt{a}$, $\sqrt{a} < z+1$ **varying** z

to the end of the instantiated schema. The **protecting** clause means that the relations $z \leq \sqrt{a}$ and $\sqrt{a} < z+1$, achieved by the loop, may not be clobbered when achieving the additional goal $z \in Z$.

The precondition for the schema's correctness is

assert $g(w,u) \leq v \equiv u \leq f(v,w)$;

instantiating it yields

assert $g(w,u) \leq v \equiv u \leq \sqrt{v}$.

Recalling that

fact $u \leq \sqrt{v} \equiv u^2 \leq v$ **when** $u \geq 0$,

this condition may be satisfied by taking $g(w,u)$ to be u^2 , provided that the argument u is never negative. This completes the analogy, obtaining the transformations

$$(f(u,v) \Rightarrow \sqrt{u}, c \Rightarrow a, e \Rightarrow 1, g(w,u) \Rightarrow u^2) .$$

Applying this instantiation to the schema, we obtain the partially written program:

```
P_s: begin comment integer square-root program
    B_s: assert $a \in N$
    achieve $z \leq \sqrt{a}$, $\sqrt{a} < z+y$ varying $z, y$
    loop L_s: assert $z \leq \sqrt{a}$, $\sqrt{a} < z+y$
        until $y \leq 1$
        $y := y/2$
        if $(z+y)^2 \leq a$ then $z := z+y$ fi
        repeat
    assert $z \leq \sqrt{a}$, $\sqrt{a} < z+1$
    achieve $z \in Z$ protecting $z \leq \sqrt{a}$, $\sqrt{a} < z+1$ varying $z$
    end .
```

That the argument $z+y$ is nonnegative follows from the invariant $\sqrt{a} < z+y$.

7. *Synthesis*

This program still contains two unachieved subgoals:

```
achieve $z \le \sqrt{a}$, $\sqrt{a} < z+y$ varying $z, y$
```

and

```
achieve $z \in Z$ protecting $z \le \sqrt{a}$, $\sqrt{a} < z+1$ varying $z$ .
```

We show now how these subgoals may be achieved.

The first subgoal is a conjunction of two relations, $z \le \sqrt{a}$ and $\sqrt{a} < z+y$, which are to be achieved simultaneously. It may be split into the two consecutive subgoals

```
purpose $z \le \sqrt{a}$, $\sqrt{a} < z+y$
    achieve $z \le \sqrt{a}$ varying $z$
    achieve $\sqrt{a} < z+y$ varying $y$
assert $z \le \sqrt{a}$, $\sqrt{a} < z+y$ .
```

The first sets the variable z to some value satisfying $z \le \sqrt{a}$; the second leaves z constant so that $z \le \sqrt{a}$ remains true, and sets y to some value satisfying $\sqrt{a} < z+y$. We can solve the first by setting

```
$z$ := 0 ;
```

since z remains 0 while achieving the next subgoal $\sqrt{a}<z+y$, that subgoal becomes

achieve $\sqrt{a}<y$ **varying** y .

We shall return to this subgoal later. Of course, there is no assurance, as yet, that this split will lead to a satisfactory solution; were it not to work out in the end, then we would have to retrace our steps to this point and try something else.

There are two ways in which we might achieve the other remaining subgoal

achieve $z\in Z$ **protecting** $z\leq\sqrt{a}$, $\sqrt{a}<z+1$ **varying** z .

One is to take the current value of z which satisfies the two conditions $z\leq\sqrt{a}$ and $\sqrt{a}<z+1$ and perturb it just enough to make it an integer while preserving those two protected relations. This can be done by assigning

$z := \lfloor z \rfloor$.

Alternatively, we note that the above subgoal is equivalent to

achieve $z\in N$ **protecting** $z\leq\sqrt{a}$, $\sqrt{a}<z+1$ **varying** z

since $z\in Z$ and $\sqrt{a}<z+1$ imply that z is nonnegative. To achieve this, we set up the new goal

achieve $z\in N$ **in** P_8 ,

by which we mean that $z\in N$ is to be a global invariant of P_8. Accordingly, we must establish $z\in N$ initially and then preserve it throughout the loop computation. Initially $z=0\in N$, as is desired. Since z is sometimes incremented by y, the latter should also be a nonnegative integer. That gives us a new goal

achieve $y\in N$ **in** P_8 .

Finally, in order to preserve the invariant $y\in N$, while it is repeatedly halved until it is no longer greater than 1, it is necessary and sufficient that $y\in 2^N$ be invariant. Thus, we have the stronger goal

achieve $y\in 2^N$ **in** P_8 .

For $y\in 2^N$ to hold throughout, we need to ensure that it holds upon entering the loop. Accordingly, we add the conjunct $y\in 2^N$ to the initialization subgoal $\sqrt{a}<y$.

We are left with the unachieved subgoal

achieve $\sqrt{a}<y$, $y\in 2^N$ **varying** y .

The first conjunct might be achieved by letting $y=a+1$, while the second could easily be achieved by letting $y=1$. However, though each conjunct is achievable by itself in this manner, achieving both together is more difficult, since in general these two solutions conflict with each other. So, instead, we transform this conjunctive goal into an iterative loop, choosing first to achieve $y \in 2^N$, and then to keep it true while executing the loop until the remaining conjunct, $\sqrt{a}<y$, is also satisfied. Since $\sqrt{\ }$ is not a primitive function, we must test for the equivalent $a<y^2$:

```
purpose $\sqrt{a}<y,\ y \in 2^N$
    achieve $y \in 2^N$ varying $y$
    loop $L_6$: assert $y \in 2^N$
        until $a<y^2$
        approach $a<y^2$ protecting $y \in 2^N$
        repeat
assert $\sqrt{a}<y,\ y \in 2^N$ .
```

To initialize $y \in 2^N$, we let $y=2^0$ and assign

$y := 1$.

Within the loop, we have the subgoal

approach $a<y^2$ **protecting** $y \in 2^N$,

i.e. we wish to preserve the invariant $y \in 2^N$ while making progress towards the exit test $a<y^2$. Since we know that initially $y=1$, and ultimately we want $0 \le \sqrt{a}<y$, it follows that y is increasing. Assuming that y is to increase monotonically, we get the loop-body subgoal

achieve $y>y_{L_6}$ **protecting** $y \in 2^N$.

where y_{L_6} is the value of y when control was last at L_6 . It follows that y must be multiplied by some positive power of 2 , e.g.

$y := 2 \cdot y$.

We have obtained the following program:

```
assert $a \in \mathbb{N}$, $z \in \mathbb{N}$, $y \in 2^{\mathbb{N}}$ in
$P_8$: begin comment integer square-root program
    $B_8$: assert $a \in \mathbb{N}$
    purpose $z \le \sqrt{a}$, $\sqrt{a} < z+y$
        $z$ := 0
        $y$ := 1
        loop $L_6$: assert $y \in 2^{\mathbb{N}}$
            until $a < y^2$
            $y$ := $2 \cdot y$
            repeat
    loop $L_8$: assert $z \le \sqrt{a}$, $\sqrt{a} < z+y$
        until $y \le 1$
        $y$ := $y/2$
        if $(z+y)^2 \le a$ then $z$ := $z+y$ fi
        repeat
    $E_8$: assert $z \le \sqrt{a}$, $\sqrt{a} < z+1$, $z \in \mathbb{N}$
    end .
```

This program can be improved as will be illustrated in the chapter on synthesis.

CHAPTER III

PROGRAM MODIFICATION AND DEBUGGING

1. INTRODUCTION

Program modification has as its goal the transformation of a given program into a new program to achieve a different goal. We have already seen how a program that divides is modified to compute square-roots. The essence of our approach is to find an *analogy* between the specifications of the given program and those of the program that we desire to construct. This analogy is then used as the basis for transforming the existing program to meet the new specifications. Invariant assertions play an important role in this process. *Program debugging* is considered as a special case of modification: if a program computes wrong results, it must be modified to achieve the intended results.

The use of analogy in problem solving in general, and theorem proving in particular, is discussed by Kling [1971]. Other works employing analogy are Brown [1976] and Chen and Findler [1976]. The modification of an already existing program to solve a somewhat different task was suggested by Manna and Waldinger [1975] as part of a program-synthesis system. Also, the STRIPS (Fikes, Hart and Nilsson [1972]) and HACKER (Sussman [1975]) systems were to some extent capable of generalizing and reusing the robot plans they generated. Recently, Ulrich and Moll [1977] have been investigating the role of analogy in program synthesis. Katz and Manna [Apr. 1975] and Sagiv [1976] *discuss debugging techniques* based on invariant assertions; Boyer, Elspas, and Levitt [1975] and King [1976] describe debugging aids based on the symbolic execution of a program.

The next section elucidates the basic aspects of our approach to program modification with the aid of several relatively straightforward examples. More subtle facets of the techniques are illustrated in the examples of Section 3. The correctness of the technique is discussed in an appendix.

2. OVERVIEW

For program modification, one is given a known correct program with its input-output specification and the specification for a desired new program; comparison of the two specifications suggests a transformation that is then applied to the given program. Even if the transformed program does not exactly fulfill the specifications, it can serve as the basis for constructing the desired new program.

1. *Basic Technique: Global Transformation*

We distinguish between two types of objects, *constants* and *variables*: a constant is any symbol appearing in a program with assumed properties, e.g. 0 , *true* , + , and ≥ ; a variable is any symbol appearing in the program with no assumed properties other than those mentioned in the input specification. A variable that changes value during program execution is termed a *program* variable; any other variable is considered to be an input variable. A program variable appearing in the output specification is called an *output* variable.

In the examples of program modification presented here, we stress transformations in which *all* occurrences of a particular symbol throughout a program are affected. Such transformations are termed "global", in contrast with "local" transformations that are applied only to a particular segment of a program.

As a simple example, consider the following annotated program (due to R.W. Floyd):

```
P1: begin comment array-minimum program
    assert n≥0
    y := n
    loop assert min(A[y:2·y])=min(A[n:2·n]) , 0≤y≤n
        until y=0
        if A[2·y-1]≤A[2·y] then A[y-1] := A[2·y-1]
                           else A[y-1] := A[2·y]
                           fi
        y := y-1
        repeat
    assert A[0]=min(A[n:2·n])
    end .
```

The symbol n appearing in the program is an input variable, A is an output variable, and y is a program variable; the symbols 0 , *min* , ≤ , etc. are constants.

Given an array segment $A[n:2 \cdot n]$ that is nonempty (i.e. n is nonnegative), when this program terminates, $A[0]$ will contain the minimum of the values of the $n+1$ array elements $A[n]$, $A[n+1]$, . . . , $A[2 \cdot n]$. This output specification is formally expressed in the final statement

assert $A[0]=min(A[n:2 \cdot n])$.

That the program satisfies this specification may be proved using the loop invariant

assert $min(A[y:2 \cdot y])=min(A[n:2 \cdot n])$, $0 \leq y \leq n$.

(This invariant holds initially when $y=n$, is maintained true by the loop body, and, together with the exit test $y=0$, implies the output specification, since $min(A[0:2 \cdot 0])=A[0]$.

To modify this program to compute the maximum of the array, rather than the minimum, we compare the specification of the given program P_1 ,

assert $A[0]=min(A[n:2 \cdot n])$,

with the output specification of the desired program P_2 ,

achieve $A[0]=max(A[n:2 \cdot n])$ **varying** $A[0:n-1]$

We say that we are looking for an analogy

$$A[0]=min(A[n:2 \cdot n]) \Leftrightarrow A[0]=max(A[n:2 \cdot n]) \ .$$

The obvious analogy between the two specifications is that one has the function *min* where the other has *max* , i.e.

$$min \Leftrightarrow max \ .$$

This analogy suggests that by replacing all occurrences of *min* in the first program, we may obtain the desired *max* program. But the transformation $min \Rightarrow max$ alone will not work. The reason is that certain properties of the constant *min* were used in the construction of the program, and those properties do not hold for the new function *max* . Later on, we shall see how this problem is dealt with.

In the meantime, there is another way to effect the transformation

$$A[0]=min(A[n:2 \cdot n]) \quad \Rightarrow \quad A[0]=max(A[n:2 \cdot n]) \ .$$

We first eliminate the function $max(A)$ by replacing it with the equivalent $-min(-A)$, where $-A$ is equal to the array A with each element negated. It remains to transform

$$A[0]=min(A[n:2 \cdot n]) \quad \Rightarrow \quad A[0]=-min(-A[n:2 \cdot n]) \ .$$

Since we do not want to transform the constant *min* , we would like the right-hand sides of both equalities to begin with *min* . Multiplying both sides of $A[0]=-min(-A[n:2 \cdot n])$ by -1 , we are left with

$$A[0]=min(A[n:2 \cdot n]) \quad \Rightarrow \quad -A[0]=min(-A[n:2 \cdot n]) \ .$$

Now the transformation

$$A \Rightarrow -A \ ,$$

applied to the output specification of the given program, yields

assert $-A[0]=min(-A[n{:}2\cdot n])$

which is equivalent to the desired

achieve $A[0]=max(A[n{:}2\cdot n])$ **varying** $A[0{:}n-1]$.

Since $A \Rightarrow -A$ is a transformation of the array variable A, and variables have no assumed properties, we can obtain a program guaranteed to satisfy the transformed specifications by applying this transformation to the program. (By applying the same transformation to all occurrences of A in the verification proof of the original program, a correctness proof for the transformed program is obtained.)

The variable A appears within the program text only in the conditional statement

```
if  $A[2\cdot y-1]\le A[2\cdot y]$  then  $A[y-1] := A[2\cdot y-1]$
                                    else  $A[y-1] := A[2\cdot y]$
                                    fi .
```

Applying the transformation $A \Rightarrow -A$ to this statement yields

```
if  $-A[2\cdot y-1]\le -A[2\cdot y]$  then  $-A[y-1] := -A[2\cdot y-1]$
                                      else  $-A[y-1] := -A[2\cdot y]$
                                      fi .
```

The test $-A[2\cdot y-1]\le -A[2\cdot y]$ is equivalent to $A[2\cdot y-1]\ge A[2\cdot y]$. But the transformed assignment statements are "illegal", since a function, in this case $-A$, may not appear on the left-hand side of an assignment. The intent of the illegal statement,

$-A[y-1] := -A[2\cdot y-1]$,

however, is for the new value of the expression $-A[y-1]$ to be made equal to the old value of $-A[2\cdot y-1]$ by changing the value of the array A. In other words, we wish to achieve the relation given by the goal

achieve $-A[y-1]=-A'[2\cdot y-1]$ **varying** $A[0{:}n-1]$,

where A' denotes the value of the array A before this **achieve** statement. To obtain an assignment to A, we must isolate the variable A on one side of the equality. We therefore multiply both sides of the equality by -1, obtaining the goal

achieve $A[y-1]=A'[2\cdot y-1]$ **varying** $A[0{:}n-1]$.

Since the variable A appears on only one side of the equality, and $0\le y-1\le n$ by virtue of the invariant $0 \le y \le n$ and the exit test $y=0$, we may achieve the desired relation between the new value of A and the old by assigning to A the value of the expression on the other side of the equality:

$$A[y-1] := A[2 \cdot y]-1 .$$

Similarly, the transformed assignment

$$-A[y-1] := -A[2 \cdot y]$$

becomes

$$A[y-1] := A[2 \cdot y] .$$

Global tranformations are applied to the invariants annotating the program as well as to the code. Thus the loop invariant

assert $min(A[y:2 \cdot y])=min(A[n:2 \cdot n])$, $0 \le y \le n$

becomes

assert $min(-A[y:2 \cdot y])=min(-A[n:2 \cdot n])$, $0 \le y \le n$,

or equivalently

assert $max(A[y:2 \cdot y])=max(A[n:2 \cdot n])$, $0 \le y \le n$.

We have derived the following program to compute the maximum:

```
P_2: begin comment array-maximum program
   y := n
   loop assert max(A[y:2·y])=max(A[n:2·n]) , 0≤y≤n
        until y=0
        if A[2·y-1]≥A[2·y] then A[y-1] := A[2·y-1]
                            else A[y-1] := A[2·y]
                            fi
        y := y-1
        repeat
   assert A[0]=max(A[n:2·n])
   end .
```

Note that the array $-A$ no longer appears in the program; only the original A is actually used.

As we have seen, global transformations are applied to all the invariants as well as to the code. In particular, a transformation that affects an input or output variable changes the output invariant correspondingly. Thus, for program modification, one looks for a transformation of input and/or output variables appearing in the output invariants that will produce invariants implying the desired output specification.

2. *Special Case: Program Debugging*

We consider the *debugging* process as an important special case of program modification: a program that computes wrong results must be modified to compute the desired (correct) results. If we know what the "bad" program actually does, then we may compare that with the specifications of what it should do, and modify (debug) the incorrect program accordingly.

As an example, consider a program intended to compute the integer square-root z of the nonnegative integer c ; that is, c should lie between the squares of the integers z and $z+1$. The goal, then, is to achieve the relation

achieve $z^2 \le c,\ \ c<(z+1)^2,\ \ z \in \mathbb{N}$,

where $\mathbb{N}$ is the set of nonnegative integers, and the given program is

```
P_s: begin comment integer square-root program
    (z,s,t) := (1,0,3)
    loop until c<s
         (z,s,t) := (z+1,s+t,t+2)
         repeat
    end .
```

Using the methods described in the chapter on annotation, invariants may be generated that express what relations this program achieves. It turns out that the global invariants

assert $t=2\cdot z+1,\ \ s=z^2-1,\ \ z \in \mathbb{N}+1$,

where $\mathbb{N}+1$ is the set of positive integers, hold throughout the program. Furthermore the loop invariant

assert $c \ge s-t$

holds whenever control is at the head of the loop. Upon termination, the global invariants, the loop invariant, and the exit test all hold:

assert $t=2\cdot z+1,\ \ s=z^2-1,\ \ z \in \mathbb{N}+1,\ \ c \ge s-t,\ \ c<s$.

It follows that the given program halts with the relations

$$\textbf{assert}\quad (z-1)^2 \le c+1,\ \ c+1 < z^2,\ \ z \in N+1$$

holding between the variables, rather than with the desired goal

$$\textbf{achieve}\quad z^2 \le c,\ \ c < (z+1)^2,\ \ z \in N\ .$$

The cause of the bug was the inadvertent exchange of the initial values of z and s.

Comparing the desired goal with the actual invariants, we note that the former may be obtained from the latter by replacing z with $z+1$ and c with $c-1$. Applying the transformation $c \Rightarrow c-1$ to the program statements affects only the exit test $c<s$, which becomes $c-1<s$, or equivalently $c \le s$. The transformation $z \Rightarrow z+1$ affects two other statements: the initialization $z:=1$ becomes $z+1:=1$ and the loop-body assignment $z:=z+1$ becomes $z+1:=z+2$. These resultant assignments, however, are "illegal", inasmuch as an expression such as $z+1$ may not appear on the left-hand side of an assignment. Instead, the expression $z+1$ is given the initial value 1 by assigning $z:=0$, and the value of the expression $z+1$ is incremented to $z+2$ by the "legal" assignment $z:=z+1$.

We have thus obtained the corrected program:

```
P_s': begin comment debugged integer square-root program
    (z,s,t) := (0,0,3)
    loop assert c≥s-t
        until c≤s
        (z,s,t) := (z+1,s+t,t+2)
        repeat
    assert z²≤c, c<(z+1)², z∈N
    end .
```

Katz and Manna [1976] suggest that when there is insufficient information to prove either the correctness or incorrectness of a program, it may nevertheless be desirable to "debug" the program. The possibly incorrect program may be transformed, using known invariants, into a new program which is unquestionably correct. Even when invariants are found for an incorrect program — it is often more difficult to discover invariants for an incorrect program since it may in fact not compute anything meaningful — there may be no way to transform them into the desired specification. It then becomes necessary to consider the sources in the code of different invariants separately.

3. *Correctness Considerations*

In the above examples, the transformed programs were correct; i.e. by the nature of the transformations, the transformed programs do in fact satisfy the transformed specifications. As we noted, this is not necessarily the case with any transformation.

Suppose, for example, that we are given the program:

```
P_4: begin comment array-minimum program
     (y,z) := (0, A[0])
     loop assert z=min(A[0:y])
          until y=n
          y := y+1
          z := min(z, A[y])
          repeat
     assert z=min(A[0:n])
     end
```

for finding the minimum of the array $A[0:n]$, and we wish to construct a program to find the maximum of the nonempty array $A[1:n]$. The given program achieves the output relation

assert $z=min(A[0:n])$,

while the output specification of the desired program is

achieve $z=max(A[1:n])$.

Thus, the transformations $min \Rightarrow max$ and $0 \Rightarrow 1$ suggest themselves. Though in this case applying these transformations happens to yield a correct program, such transformations of constant symbols do not necessarily preserve correctness. Were the function *min* not explicitly used in the program, e.g. in the program

```
P_4': begin comment alternative array-minimum program
      (y,z) := (0, A[0])
      loop assert z=min(A[0:y])
           until y=n
           y := y+1
           if A[y]<z then z := A[y] fi
           repeat
      assert z=min(A[0:n])
      end ,
```

then the proposed transformation $min \Rightarrow max$ would clearly not work.

Global transformations, where an input variable is systematically replaced by a function of only input variables, or an output variable by a function of output variables or of both input and output variables (as in the previous examples), always yield a program satisfying the transformed specifications. However, transformations of constant symbols (as in this last example) are not guaranteed to result in a program satisfying the specifications. Details regarding correctness-preserving transformations may be found in an appendix.

Hence, for some transformations, correctness must be verified. For this purpose, invariant assertions are utilized. As we saw above, invariants are essential in our approach to debugging too, as it is necessary to have some idea of what the program actually does before it can be corrected.

Global transformations are applied to all invariants, as well as to the code. Using these transformed invariants, verification conditions for the new program may be generated; if they hold, then the new program is correct. Alternatively, applying the transformations to the (unsimplified) verification conditions of the original program yields verification conditions for the transformed program. It is best if the conditions are given in the form of a subgoal tree, reflecting the logical steps taken in constructing the program. This subgoal structure may be expressed in **purpose** statements.

Returning to the above example, we wish to modify the first version of the *min* program P_4, to obtain a program that achieves $z=max(A[1:n])$. Applying the transformations $min \Rightarrow max$ and $0 \Rightarrow 1$ yields

```
P_8: begin comment array-maximum program
    (y, z) := (1, A[1])
    loop assert z=max(A[1:y])
         until y=n
         y := y+1
         z := max(z, A[y])
         repeat
    assert z=max(A[1:n])
    end .
```

Using the new invariants, the correctness of this *max* program may straightforwardly be shown.

On the other hand, applying these transformations to the alternative *min* program P_4' would yield

```
P_5': begin comment suggested array-maximum program
      (y, z) := (0, A[0])
      loop suggest z=max(A[0:y])
           until y=n
           y := y+1
           if A[y]<z then z := A[y] fi
           repeat
      suggest z=max(A[0:n])
      end .
```

We have replaced the assertions with suggestions, since correctness is not guaranteed by transformations that involve constants (in our case *min*). Indeed this program is incorrect, since the loop body does not preserve the candidate loop invariant

suggest $z=max(A[0:y])$.

In the next subsection, we discuss what can be done in such cases.

4. *Completing an Analogy*

As discussed above, the verification conditions will not always hold for a given set of transformations. There could, for example, be unrelated occurrences of 0 in the *min* program P_4' (in which case the global transformation $0 \Rightarrow 1$ would be inappropriate) or the function symbol *min* might not appear explicitly in the program at all (and therefore the transformation *min* ⇒ *max* would be ineffectual).

The program

```
P_4': begin comment alternative array-minimum program
      (y, z) := (0, A[0])
      loop assert z=min(A[0:y])
           until y=n
           y := y+1
           if A[y]<z then z:= A[y] fi
           repeat
      assert z=min(A[0:n])
      end
```

has several verification conditions. One of them corresponds to the loop-body path when the conditional test is true; it may be represented as

```
assert $y \neq n$
$y$ := $y+1$
assert $A[y]<z$
$z$ := $A[y]$ .
```

Since the program is correct, we know that the loop invariant holds each time control is at the head of the path, and that if that path is taken, then the invariant holds at the end of the path as well. So, assuming that the invariant $z=min(A[0:y])$ holds and that the exit test $y=n$ is false, then after incrementing y and setting z to $A[y]$ when the conditional test $A[y]<z$ is true, the invariant $z=min(A[0:y])$ again holds, for the new values of y and z . In other words, we have

$$z=min(A[0:y]) \wedge y \neq n \wedge A[y+1]<z \supset A[y+1]=min(A[0:y+1]) .$$

Applying the two transformations, $0 \Rightarrow 1$ and $min \Rightarrow max$, to this condition we obtain

$$z=max(A[1:y]) \wedge y \neq n \wedge A[y+1]<z \supset A[y+1]=max(A[1:y+1]) .$$

However, the condition no longer holds, and we must try to find a way to correct that. The condition is equivalent to

$$z=max(A[1:y]) \wedge y \neq n \wedge A[y+1]<z \supset A[y+1]=max(max(A[1:y]), A[y+1]) ,$$

which, in turn, simplifies to

$$A[y+1]<z \supset A[y+1]=max(z, A[y+1]) ,$$

or

$$A[y+1]<z \supset A[y+1]\geq z .$$

Now, matching the two sides of this implication, suggests completing the analogy with the additional transformation $< \Rightarrow \geq$.

This transformation in fact makes all the verification conditions valid and yields a correct program for finding the maximum:

```
P₈'': begin comment alternative array-maximum program
    (y, z) := (1, A[1])
    loop assert z=max(A[1:y])
        until y=n
        y := y+1
        if A[y]≥z then z := A[y] fi
        repeat
    assert z=max(A[1:n])
    end .
```

Note that the additional transformation could be localized to the conditional statement, since its verification conditions are the only ones that fail.

Were 0 not to appear explicitly in the initialization, say if we had instead

y := 1-1
z := $A[y]$,

then the verification condition for this path — after applying the transformation $(0 \Rightarrow 1, min \Rightarrow max)$ — would be

$$A[1\text{-}1]=max(A[1:1\text{-}1]) \ ,$$

which does not hold. It may then be necessary to write a new program segment that would initialize the loop invariant $z=max(A[1:y])$ by setting y and z to appropriate values. The goal

achieve $z=max(A[1:y])$ **varying** y, z

can be satisfied by the assignment

(y, z) := $(1, A[1])$.

Were 0 to appear in unrelated parts of the program — say for the purpose of illustration that we had an additional loop-body assignment y:=y+0 — then the transformation $0 \Rightarrow 1$ would result in an incorrect program. In such cases, analysis of the (loop-body) verification conditions would suggest not applying that transformation to that occurrence of 0 .

Another problem that sometimes arises in program modification is that the transformations only achieve part of the output specification. In such cases, it may be possible to extend the program to achieve all the desired parts by achieving the missing parts at the onset and maintaining them invariantly true until program termination.

Alternatively, we could append new code to the end of the program that will achieve the additional parts — without "clobbering" what has already been achieved by the program.

For example, consider the case where it is desired that P_5'' also find the position x, in the array, of the minimum element z. We can extend the program to

achieve $z=A[x]$ **in** P_5'' **varying** x

by maintaining that relation as an invariant throughout the execution of the program.

Synthesis and extension are treated in a separate chapter.

3. EXAMPLES

In this section, we present two examples of program modification. We begin with an incorrect real-division program and show how to correct it. This is the same program as appeared in the overview chapter; here we go into greater detail. Then we modify a square-root program to search a sorted array for a particular element.

Example 1: *Bad Real Division to Good Real Division.*

Consider the problem of computing the quotient z of two nonnegative real numbers c and d, where $c<d$, within a specified tolerance ϵ, $0<\epsilon$. The given program is:

```
P6: begin comment bad real-division program
    assert $0\le c<d,\ 0<\epsilon$
    $(z,y) := (0,1)$
    loop suggest $z\le c/d,\ c/d<z+y$
         until $y\le\epsilon$
         if $d\cdot(z+y)\le c$ then $z := z+y$ fi
         $y := y/2$
         repeat
    suggest $z\le c/d,\ c/d<z+\epsilon$
    end .
```

The initial assertion

assert $0 \le c < d,\ 0 < e$

contains the input specification that the input variables c, d and e are assumed to satisfy. The statement

suggest $z \le c/d,\ c/d < z+e$

at the end of the program expresses the output specification of the program which the program is believed to achieve. But, for example, $c=1$, $d=3$, and $e=1/3$, which satisfy the input specification, yield $z=0$ which does not satisfy the second conjunct $c/d<z+e$ of the output specification.

Before we can debug this program, we must know more about what it actually does. For this purpose, we first annotate the program with *loop* and *output invariants*. The annotated program — with invariants that correctly express what the program does — is:

```
assert $0 \le c < d,\ 0 < e$
$(z, y)$ := $(0, 1)$
loop assert $d \cdot z \le c,\ c < d \cdot (z+2 \cdot y)$
     until $y \le e$
     if $d \cdot (z+y) \le c$ then $z$ := $z+y$ fi
     $y$ := $y/2$
     repeat
assert $d \cdot z \le c,\ c < d \cdot (z+2 \cdot e)$ .
```

The desired relation $c/d<z+e$ is *not* implied by the output invariants.

We now have the task of finding a transformation (correction) that transforms the actual output invariant

assert $d \cdot z \le c,\ c < d \cdot (z+2 \cdot e)$

into the desired goal

suggest $z \le c/d,\ c/d < z+e$,

or equivalently,

suggest $d \cdot z \le c,\ c < d \cdot (z+e)$.

The transformation will then be applied to the program. Accordingly, we would like to modify the program in such a manner as to transform the insufficiently strong $c<d\cdot(z+2\cdot e)$ into the desired $c<d\cdot(z+e)$ and at the same time preserve the correctness of the other conjunct of the specification:

$$(c<d\cdot(z+2\cdot e) \Rightarrow c<d\cdot(z+e), d\cdot z\leq c \Rightarrow d\cdot z\leq c) \ .$$

The expressions $c<d\cdot(z+2\cdot e)$ and $c<d\cdot(z+e)$ differ in that the former has $2\cdot e$ where the latter has just e. So if we can transform $2\cdot e \Rightarrow e$, then we will have transformed the specifications as desired. In order to transform the expression $2\cdot e$ into e, we can transform the input variable e into $e/2$. We, therefore, apply the transformation

$$e \Rightarrow e/2$$

to all occurrences of e in the program; all other symbols in the program are left unchanged. Only one executable statement — the exit clause — is affected, giving

Correction 1: *Replace the exit clause with*
until $y\leq e/2$.

The resulting program is:

```
P6': begin comment corrected real-division program
    assert 0≤c<d, 0<e/2
    (z,y) := (0,1)
    loop assert d·z≤c, c<d·(z+2·y)
         until y≤e/2
         if d·(z+y)≤c then z := z+y fi
         y := y/2
         repeat
    assert d·z≤c, c<d·(z+2·e/2) .
```

Had we matched the output invariants

assert $d \cdot z \le c,\ c < d \cdot (z+2 \cdot e)$

with the original output specification

suggest $z \le c/d,\ c/d < z+e$,

then the transformations

$(d \Rightarrow 1, c \Rightarrow c/d, e \Rightarrow e/2)$

would suggest themselves. Here d is transformed into the identity element of multiplication, so that $d \cdot z \Rightarrow z$. This set of transformations leads to the same program, except for the fact that the conditional test is transformed into $z+y \le c/d$ which contains the nonprimitive division operator. Since d is positive, this nonprimitive test is equivalent to the primitive test $d \cdot (z+y) \le c$, which may be substituted for it.

Two additional debugging transformations may be obtained. Again we begin by comparing $c < d \cdot (z+2 \cdot e)$ with $c < d \cdot (z+e)$, but this time we try to leave e unchanged. We therefore try to isolate e on the right of both inequalities. Accordingly, we wish to transform

$(c/d-z)/2 < e \quad \Rightarrow \quad c/d-z < e$.

Matching the two sides of the inequalities leaves us with $(c/d-z)/2 \Rightarrow c/d-z$. Multiplying both by 2 , we get

$c/d-z \quad \Rightarrow \quad 2 \cdot (c/d-z) = 2 \cdot c/d-2 \cdot z = c/(d/2)-2 \cdot z$.

This leads to the transformations

$(c \Rightarrow 2 \cdot c, z \Rightarrow 2 \cdot z)$

or

$(d \Rightarrow d/2, z \Rightarrow 2 \cdot z)$.

Applying these transformations to the second conjunct $d \cdot z \le c$ gives either $d \cdot 2 \cdot z \le 2 \cdot c$ or $d/2 \cdot 2 \cdot z \le c$, both of which simplify to $d \cdot z \le c$. This is exactly what was wanted and no further transformations are necessary.

Doubling z and either doubling c or halving d in the conditional test $d \cdot (z+y) \le c$ yields a test equivalent to $d \cdot (z+y/2) \le c$. Transforming z into $2 \cdot z$ affects two additional statements: the initialization $z := 0$ becomes the "illegal" assignment $2 \cdot z := 0$, which is equivalent to the "legal"

$z := 0$.

Similarly, the assignment $z := z+y$ of the **then**-branch becomes $2 \cdot z := 2 \cdot z+y$; in order to

achieve $2\cdot z=2\cdot z'+y$ **varying** z ,

we can assign

$z := z+y/2$.

No other statements are affected by either of the two modifications; thus they both yield:

Correction 2: *Replace the conditional statement with*
if $d\cdot(z+y/2)\le c$ **then** $z := z+y/2$ **fi** .

In general: In order to transform $f(u) \Rightarrow v$, for any expressions u and v and function f, we may transform $u \Rightarrow f^{-}(v)$, where f^{-} is the inverse of f. When applying a transformation $y \Rightarrow f(y)$ to an assignment $y:=g(y)$, we get an illegal assignment

$f(y) := g(f(y))$.

To

achieve $f(y)=g(f(y'))$ **varying** y ,

we can apply the inverse function f^{-} to both sides, suggesting the assignment

$y := f^{-}(g(f(y)))$.

Each of these possible sets of transformations involved one of the input variables e, c, or d. One must, however, be careful when transforming input variables, since the transformation should be applied to the input assertion as well, possibly changing the range of legal inputs thereby. In our case, the transformations we have performed pose no problem: Applying $e \Rightarrow e/2$ to the input assertion

assert $0\le c<d,\ 0<e$

yields the equivalent assertion

assert $0\le c<d,\ 0<e/2$.

Therefore, halving e has no effect on the input range, and the transformed program is correct for any inputs satisfying the given specification. Moreover, since in fact the condition $c<2\cdot d$, rather than $c<d$, is strong enough to imply that the loop invariants $d\cdot z\le c$ and $c<d\cdot(z+2\cdot y)$ hold after the initialization assignment $(z,y):=(0,1)$, (this is easily seen by substituting 0 and 1 for z and y, respectively, in the invariants), we can relax the input assertion of P_6 to

assert $0\le c<2\cdot d,\ 0<e$.

Then, replacing the c in $c<2 \cdot d$ by $2 \cdot c$ (or the d by $d/2$) still yields a program correct for inputs satisfying $c<d$, as is desired.

Our program after Correction 2, annotated with appropriately modified invariants is (all c have been replaced by $2 \cdot c$ and all z by $2 \cdot z$ and the resultant expressions have been simplified):

```
P6'': begin comment good real-division program
      assert 0≤c<d, 0<e
      (z,y) := (0,1)
      loop assert d·z≤c, c<d·(z+y)
           until y≤e
           if d·(z+y/2)≤c then z := z+y/2 fi
           y := y/2
           repeat
      assert d·z≤c, c<d·(z+e)
      end .
```

Example 2: *Real Square-root to Array Search.*

In this example, we show how the square-root program

```
P7: begin comment square-root program
    assert a≥1, e>0
    (z,y) := (1,a-1)
    loop assert z≤√a, √a<z+y
         until y≤e
         y := y/2
         if (z+y)²≤a then z := z+y fi
         repeat
    assert z≤√a, √a<z+e
    end
```

may be modified to obtain a program that searches for the position z of an element b known to occur in an array segment $A[1:n]$. The array is assumed to contain nonnegative integers sorted in nondescending order. This example will illustrate a number of difficulties that may be encountered.

Our goal is

```
P_8: begin comment array-search program
       assert $u \le v \supset A[u] \le A[v]$, $A[u] \in \mathbb{N}$, $b \in bag(A[1:n])$
       achieve $A[z]=b$ varying $z$
       end ,
```

where $\mathbb{N}$ is the set of nonnegative integers and $bag(A[1:n])$ denotes the multiset (bag) of elements in the array segment $A[1:n]$. We shall allow indexing of an array by any real number, and adopt the convention that the intended element may be found by truncating the index, i.e.

fact $A[u]=A[\lfloor u \rfloor]$.

(In a similar manner, we could develop a program following the Algol-60 convention of rounding-off the index.)

The desired goal

achieve $A[z]=b$ **varying** z

is not directly comparable with the output invariants of the given program

assert $z \le \sqrt{a}$, $\sqrt{a} < z+e$.

So we first develop the goal somewhat.

As a first try, we replace the desired goal with the equivalent conjunctive goal

achieve $A[z] \le b$, $b \le A[z]$ **varying** z ,

guided by the fact that we wish to achieve an equality, while the given program achieves an inequality. Since we are dealing with integers, this is the same as

achieve $A[z] \le b$, $b < A[z]+1$ **varying** z .

Accordingly, we are looking for a transformation

$z \le \sqrt{a} \wedge \sqrt{a} < z+e \Rightarrow A[z] \le b \wedge b < A[z]+1$.

and try to compare the conjunct $z \le \sqrt{a}$ with $A[z] \le b$. (Since $\wedge$ is commutative, we could just as well begin by trying to compare $z \le \sqrt{a}$ with $b < A(z)+1$.) Matching the two sides of the inequality, we get $(z \Rightarrow A[z], \sqrt{a} \Rightarrow b)$; in order to obtain $\sqrt{a} \Rightarrow b$, we can let $a \Rightarrow b^2$. Applying these transformations to the remaining conjunct $\sqrt{a} < z+e$ leaves us with $b < A[z]+e \Rightarrow b < A[z]+1$; this suggests the additional transformation $e \Rightarrow 1$.

Applying the three transformations

$$(z \Rightarrow A[z], a \Rightarrow b^2, e \Rightarrow 1)$$

to the given square-root program yields

```
P_6: begin comment proposed array-search program
     assert $b^2 \ge 1$, $1>0$
     $(A[z], y)$ := $(1, b^2-1)$
     loop assert $A[z] \le b$, $b < A[z]+y$
          until $y \le 1$
          $y$ := $y/2$
          if $(A[z]+y)^2 \le b^2$ then $A[z]$ := $A[z]+y$ fi
          repeat
     assert $A[z] \le b$, $b < A[z]+1$
     end .
```

There are, however, a number of problems with this program, the insurmountable one lying in the conditional-branch assignment $A[z]{:=}A[z]+y$. The problem is that the original goal stated that only z is an output variable, while the array A is an input variable which may not be modified by an assignment. Furthermore, there is *no* way to

achieve $A[z]=A[z']+y$ **varying** z ,

since the value $A[z']+y$ is not known to appear in A at all.

So we must look for another alternative. Since $A[u]=A[\lfloor u \rfloor]$, it is sufficient to

achieve $A[\lfloor z \rfloor]=b$ **varying** z

in order to achieve our goal

achieve $A[z]=b$ **varying** z .

At this point, we would like to extract z from within the expression $A[\lfloor z \rfloor]$, as it appears by itself in the output invariants of the given program. To this end, we use the function $pos(A, u)$ which gives the position of the (rightmost) occurrence of the element u in the array A ; it is an inverse of the array indexing function $A[v]$, i.e.

fact $pos(A,u) \in Z$ **when** $u \in A$

(Z is the set of all integers),

fact $A[pos(A,u)]=u$ **when** $u \in A$,

and

fact $pos(A,u) > v-1$ **when** $A[v]=u$.

Instantiating the second fact with b for u yields $A[pos(A,b)]=b$, since it is given that $b \in A$; thus, in order to

achieve $A[\lfloor z \rfloor]=b$ **varying** z ,

it suffices to

achieve $pos(A,b)=\lfloor z \rfloor$ **varying** z .

Applying now the definition of $\lfloor u \rfloor$,

fact $v=\lfloor u \rfloor \equiv v \leq u \wedge u < v+1 \wedge v \in Z$,

we obtain the conjunctive goal

achieve $pos(A,b) \leq z$, $z < pos(A,b)+1$, $pos(A,b) \in Z$ **varying** z .

Since the third conjunct $pos(A,b) \in Z$ is always true, we are left with the goal

achieve $pos(A,b) \leq z$, $z < pos(A,b)+1$ **varying** z .

The current goal is still not readily comparable with the output specification of the real square-root program,

assert $z \leq \sqrt{a}$, $\sqrt{a} < z+\epsilon$:

while for the array-search program the output variable z appears on the right-hand side of the $\leq$ relation and on the left-hand side of the $<$ relation, for the square-root program the sides are reversed.

One possible solution is to transform the predicates $\leq$ and $<$. To get

$z \leq \sqrt{a} \Rightarrow z \geq pos(A,b)$,

we may apply the transformations

$(\leq \Rightarrow \geq, \sqrt{a} \Rightarrow pos(A,b))$.

To obtain the second transformation $\sqrt{a} \Rightarrow pos(A,b)$, we let $a \Rightarrow pos(A,b)^2$. Applying these transformations to the conjunct $\sqrt{a}<z+e$ leaves

$$pos(A,b)<z+e \Rightarrow pos(A,b)+1>z \ .$$

Transposing to get just $pos(A,b)$ on the left of the inequalities, gives

$$pos(A,b)<z+e \Rightarrow pos(A,b)>z-1 \ ,$$

so we add the transformations $(< \Rightarrow >, e \Rightarrow -1)$. All together we have

$$(\leq \Rightarrow \geq, a \Rightarrow pos(A,b)^2, < \Rightarrow >, e \Rightarrow -1) \ .$$

Applying these transformations to the given square-root program yields

```
(z,y) := (1, pos(A,b)²-1)
loop suggest z≥pos(A,b), pos(A,b)>z+y
     until y≥-1
     y := y/2
     if (z+y)²≥pos(A,b)² then z := z+y fi
     repeat
suggest z≥pos(A,b), pos(A,b)>z-1 .
```

Simplifying the expressions in the program, we get

```
(z,y) := (1, pos(A,b)²-1)
loop suggest z≥pos(A,b), pos(A,b)>z+y
     until y≥-1
     y := y/2
     if z+y≥pos(A,b) then z := z+y fi
     repeat
suggest z≥pos(A,b), pos(A,b)>z-1 .
```

Before we try to eliminate the nonprimitive function pos from the transformed program, we attempt to verify the correctness of the program as is. The loop invariants

assert $z \geq pos(A,b),\ pos(A,b)>z+y$

along with the exit condition

until $y \geq -1$

clearly imply the desired output invariant

assert $pos(A,b) \leq z,\ z<pos(A,b)+1$.

Furthermore, the loop-body path preserves the loop invariants for both cases of the conditional.

The problem is with the verification condition for the initialization path: the assignment $(z,y):=(1,pos(A,b)^2-1)$ does not initialize the loop invariants. So we replace the initialization with the new subgoal

```
assert  $u \le v \supset A[u] \le A[v]$,  $A[u] \in N$,  $b \in bag(A[1:n])$
achieve  $z \ge pos(A,b)$,  $pos(A,b) > z+y$  varying  $z, y$ .
```

Since we are given that b appears within the segment $A[1:n]$, we can achieve the relation $z \ge pos(A,b)$ by letting $z=n$. Now we can achieve $pos(A,b) > z+y$ by insisting that $z+y=0$, for which we initialize y to $-z=-n$.

The verification condition for termination is $(\exists \xi \in N)(-n/2^{\xi} \ge -1)$, i.e. by repeatedly halving y, which has the initial value $-n$, the exit test $y \ge -1$ must at some point become true. This is indeed the case. Thus, all the verification conditions hold and the transformed program is correct.

Finally, the conditional test $z+y \ge pos(A,b)$, that contains the nonprimitive function pos, may be replaced by $A[z+y+1] > b$. That the two tests are equivalent, may be deduced from the input specification

```
assert  $u \le v \supset A[u] \le A[v]$
```

and the definition of pos.

Our program now looks like this:

```
(z,y) := (n,-n)
loop assert  $z \ge pos(A,b)$,  $pos(A,b) > z+y$
     until  $y \ge -1$
     y := y/2
     if  $A[z+y+1] > b$  then  z := z+y  fi
     repeat
assert  $z \ge pos(A,b)$,  $pos(A,b) > z-1$ .
```

If we transform the program variable y by $y \Rightarrow -y$ and simplify, we get

```
P_8: begin comment array-search program
     assert $u\le v \supset A[u]\le A[v]$, $A[u]\in\mathbb{N}$, $b\in bag(A[1:n])$
     $(z,y)$ := $(n,n)$
     loop assert $z\ge pos(A,b)$, $pos(A,b)>z-y$
          until $y\le 1$
          $y$ := $y/2$
          if $A[z-y+1]>b$ then $z$ := $z-y$ fi
          repeat
     assert $A[z]=b$
     end .
```

Note that transforming a variable that does not appear in the program specifications, such as y, cannot affect *what* the program does, only *how* it does it.

Having found one satisfactory solution, let us return to the point where we compared the desired goal

achieve $z\ge pos(A,b)$, $pos(A,b)+1>z$ **varying** z

with the output invariants

assert $z\le\sqrt{a}$, $\sqrt{a}<z+e$.

An alternative way to transform the relation $\le$ in the desired goal, into $\ge$ and $<$ into $>$, without transforming the predicates themselves, would be to multiply both sides of the inequalities by -1. We would thereby obtain the equivalent goal

achieve $-z\le -pos(A,b)$, $-pos(A,b)-1<-z$ **varying** z .

Comparing the output invariant $z\le\sqrt{a}$ with the first conjunct of this goal, suggests the transformations

$(z \Rightarrow -z, \sqrt{a} \Rightarrow -pos(A,b))$.

To obtain $\sqrt{a}\Rightarrow -pos(A,b)$, we would like to use $a \Rightarrow (-pos(A,b))^2$, but since $pos(A,b)$ is positive, that would give $\sqrt{a}\Rightarrow pos(A,b)$, rather than $-pos(A,b)$ as desired. As there is no easy way out of this problem, we drop this possibility.

There is another way: Just as $u\le v$ is equivalent to $u<v+1$ for integers u and v, for real numbers we can transform an expression of the form $u\le v$ into one of form $u<v+\epsilon$, where ϵ is an arbitrarily small real number. Similarly $u<v$ is equivalent to $u+\epsilon\le v$. The ϵ's may then be eliminated from the resulting program. Thus, we may compare

assert $z<\sqrt{a}+\epsilon$, $\sqrt{a}<z+e$

with

achieve $z<pos(A,b)+1,\ \ pos(A,b)<z+\epsilon$,

which suggests the set of transformations

$(a \Rightarrow pos(A,b)^2,\ z \Rightarrow z-1+\epsilon,\ e \Rightarrow 1)$.

Applying these transformations to the square-root program yields

```
assert $pos(A,b)^2 \geq 1,\ 1>0$
$(z-1+\epsilon, y)$ := $(1, pos(A,b)^2-1)$
loop assert $z-1+\epsilon \leq pos(A,b),\ pos(A,b)<z-1+\epsilon+y$
     until $y \leq 1$
     $y$ := $y/2$
     if $(z-1+\epsilon+y)^2 \leq pos(A,b)^2$ then $z-1+\epsilon$ := $z-1+\epsilon+y$ fi
     repeat
assert $z-1+\epsilon \leq pos(A,b),\ pos(A,b)<z-1+\epsilon+1$ .
```

The transformed conditional test $(z-1+\epsilon+y)^2 \leq pos(A,b)^2$, may be simplified to $z-1+\epsilon+y \leq pos(A,b)$, i.e. $z-1+y<pos(A,b)$. To remove the nonprimitive function pos, we replace the test with $A[z+y] \leq b$. Replacing the initialization and the illegal assignments, we obtain the transformed program:

```
$P_6'$: begin comment array-search program
    assert $u \leq v \supset A[u] \leq A[v],\ A[u] \in N,\ b \in bag(A[1:n])$
    $(z, y)$ := $(1, n)$
    loop assert $z<pos(A,b)+1,\ pos(A,b)+1 \leq z+y$
         until $y \leq 1$
         $y$ := $y/2$
         if $A[z+y] \leq b$ then $z$ := $z+y$ fi
         repeat
    assert $A[z]=b$
    end .
```

Replacing the initialization in general requires rechecking the verification condition for termination; in this case, $(\exists \zeta \in N)(n/2^{\zeta} \leq 1)$ must hold for the program to terminate, as indeed it does.

This array-search program may be given a more conventional appearance if we can replace $z+y$ (the right bound of the search), which appears twice in the program, with just y. To effect $z+y \Rightarrow y$, we use the global transformation

$$y \Rightarrow y-z ,$$

since $z+(y-z)=y$. Since the right-hand side of this transformation contains a variable other than the transformed variable y, the application of the transformation is a bit trickier:

wherever there is an assignment to z, even if y is not changed, we must consider what happens to the transformed value of y. Thus, the conditional branch assignment $z:=z+y$ must be considered as though it were $(z,y):=(z+y,y)$, which is transformed into $(z,y-z):=(z+(y-z),y-z)$.

Replacing all occurrences of y in the program with $y-z$, and using the appropriate **achieve** statements in place of the transformed assignments, we get:

```
achieve $z=1,\ y-z=n$ varying $z,y$
loop assert $z<pos(A,b)+1,\ pos(A,b)+1\le y$
     until $y-z\le 1$
     achieve $y-z=(y'-z)/2$ varying $y$
     if $A[y]\le b$ then achieve $z=z'+(y'-z'),\ y-z=y'-z'$ varying $z,y$ fi
     repeat
assert $A[z]=b$ .
```

The initialization subgoal

achieve $z=1,\ y-z=n$ **varying** z,y

yields the assignment

$(z,y) := (1,n+1)$.

Isolating y on one side of the equality in the subgoal

achieve $y-z=(y'-z)/2$ **varying** y ,

yields

$y := (z+y)/2$.

The conditional-branch subgoal

achieve $z=z'+(y'-z'),\ y-z=y'-z'$ **varying** z,y

yields the assignment

$(z,y) := (y,2\cdot y-z)$.

The program, so far, is:

```
(z,y) := (1,n+1)
loop assert z<pos(A,b)+1,  pos(A,b)+1≤y
     until y-z≤1
     y := (z+y)/2
     if A[y]≤b then (z,y) := (y,2·y-z) fi
     repeat
assert A[z]=b .
```

There are still a few more changes that may be made: By pushing the loop-body assignment to y into the conditional statement, we get

```
if A[(z+y)/2]≤b then (z,y) := ((z+y)/2,y) else y := (z+y)/2 fi .
```

Now, by eliminating the superfluous assignment $y:=y$ and introducing a temporary variable t to contain the value $(z+y)/2$, we get

```
t := (z+y)/2
if A[t]≤b then z := t else y := t fi .
```

Our final version of the array-search program is:

```
P₂': begin comment conventional array-search program
     assert u≤v⊃A[u]≤A[v],  A[u]∈N,  b∈bag(A[1:n])
     (z,y) := (1,n+1)
     loop assert z<pos(A,b)+1,  pos(A,b)+1≤y
          until y-z≤1
          t := (z+y)/2
          if A[t]≤b then z := t else y := t fi
          repeat
     assert A[z]=b
     end .
```

This chapter has illustrated the use of analogy to modify and debug programs. The next chapter shows how similar techniques may be used to abstract and instantiate programs.

CHAPTER IV

PROGRAM ABSTRACTION AND INSTANTIATION

1. INTRODUCTION

When confronted with a new task, a person will often notice a resemblance between it and some previous accomplished task. To conserve effort, he is likely to adapt the knowledge he learned on those occasions to the new problem at hand. After solving a number of similar.problems, he might form a general paradigm for solving such problems by supressing the inconsequential particulars of the individual instances. We term the process of forming a general scheme from instances of the problem *abstraction*, and that of applying a general scheme to a particular problem *instantiation*.

In the programming case, we are given a set of concrete programs, presumably related in some way, and would like to derive an abstract program schema. The programs are assumed to be annotated with their input-output specifications and with sufficient invariant assertions to demonstrate their correctness. The first step is to find an abstraction of the set of specifications of all the programs. This yields an abstract specification that may be instantiated to any of the given concrete specifications. For each of the given specifications there corresponds an abstraction mapping that when applied to the concrete specification will yield the abstract specification. That same mapping, applied to the given program, yields an abstract program schema. Conversely, the instantiation mapping that yields the concrete specifications of a program when applied to the abstract specifications, will yield the corresponding concrete program when applied to the abstract schema.

A schema, however, may not be applicable to all possible instantiations of its specifications. In that case, the schema is accompanied by an input specification containing conditions that must be satisfied by the instantiation to guarantee correctness. These preconditions may be derived from the verification conditions which serve to bridge the gap between the assertion language in which the specifications are stated and the programming language in which the program is coded. In cases where the preconditions are not satisfied by a particular instantiation, analysis of the unsatisfied conditions may suggest modifications that will help satisfy the conditions.

To date, little research has been done on program abstraction. The STRIPS system (Fikes, Hart and Nilsson [1972]) generalized the loop-free robot plans it generated; the HACKER system (Sussman [1975]) "subroutinized" and generalized the "blocks-world" plans it generated, but was limited in this respect by its use of executions, rather than verification proofs, to determine what program constants could be generalized. Recently, Gerhart [Apr. 1975] and Gerhart and Yelowitz [1976] have also advocated the use of program schemata as a powerful programming tool and have recommended the hand-compilation of a handbook of such program schemata to aid human programmers.

(See Yelowitz and Duncan [1977] for a detailed example of the use of schemata as an aid in program verification and Misra [1978] for an approach to the specification of schemata.)

In this chapter, we apply program-modification techniques to abstraction. To generalize the common aspects of two programs P and Q and form a program schema, we first find the set of transformations for modifying P into Q. Recall that to modify a program P so as to obtain a program Q, we look for an analogy $P \Leftrightarrow Q$ between the specifications of P and Q. The analogy suggests a set of transformations that when applied to the program P will yield a program satisfying the specifications of Q. Each transformation is of the form $e \Rightarrow f$; all occurrences of e in the program P are transformed into f. For each such transformation, the corresponding abstraction transformation for P is $e \Rightarrow v$ and for Q is $f \Rightarrow v$, where v is a new variable symbol.

We are tacitly assuming that the specifications of our programs are expressed formally and that this specification does express the desired behavior of the program. This is not a trivial point; it is not uncommon for errors or omissions to be made in the original specification of a program. Balzer, Goldman, and Wile [1977] have been investigating the possibility of constructing a formal specification from informal and incomplete specifications.

Another problem inherent in our use of analogy for program modification and abstraction, is that the two specifications that are to be compared may have little syntactically in common. When the specifications are not syntactically similar, it is necessary to rephrase the given specifications in some equivalent manner that brings their similarity to the fore. This is clearly a difficult problem. In our examples, we indicate what may be done in some such cases; in general, some form of means-end analysis seems appropriate.

In the next section, we present several examples of abstraction and instantiation.

2. EXAMPLES

Three examples follow: in the first, we derive a schema for linear search; the second is an iterative implementation of a recursive definition; the third is a more general binary search than the one we saw in Chapter II.

Example 1: *Minimum/Maximum Schema.*

Consider the following two programs:

```
P₁: begin comment minimum-value program
      assert n∈N
      (z,y) := (A[0],0)
      loop assert z≤A[0:y], y∈N
          until y=n
          y := y+1
          if A[y]<z then z := A[y] fi
          repeat
      assert z≤A[0:n]
      end
```

and

```
Q₁: begin comment maximum-position program
      assert m,n∈Z, m≤n
      (z,y) := (n,n)
      loop assert A[z]≥A[y:n], y∈Z
          until y=m
          y := y-1
          if A[y]>A[z] then z := y fi
          repeat
      assert A[z]≥A[m:n]
      end ,
```

where the construct $p[u{:}v]$ is shorthand for $(\forall\zeta)(u\le\zeta\le v)p(\zeta)$, for any predicate p, i.e., p holds for all values ζ in the range [u:v]. The output specification of the first program P_1 is

assert $z\le A[0{:}n]$;

it finds a value z smaller than any appearing in the array segment $A[0{:}n]$. The second program Q_1 finds the position z of a maximum element in the array segment $A[m{:}n]$; its output specification is

assert $A[z]\ge A[m{:}n]$.

(For simplicity we have not included the output specifications $z\in A[0{:}n]$ and $m\le z\le n$ of

P_1 and Q_1, respectively.)

In the previous chapter we saw how one may derive such programs one from the other. The obvious analogy between the specifications of the two programs is that where the specifications of P_1 have $\leq$, z, and 0, the specifications of Q_1 have $\geq$, $A[z]$, and m, respectively:

$$(\leq \Leftrightarrow \geq, z \Leftrightarrow A[z], 0 \Leftrightarrow m) \ .$$

Applying the transformations

$$(\leq \Rightarrow \geq, z \Rightarrow A[z], 0 \Rightarrow m)$$

to the output specification of P_1 transforms it into that of Q_1. However, applying these transformations to the program P_1 does not yield a program satisfying the specifications of Q_1. This is because the program P_1 makes use of properties of the constant $\leq$ that do not apply to $\geq$. Similarly, applying the transformations

$$(\geq \Rightarrow \leq, z \Rightarrow pos(A,z), m \Rightarrow 0)$$

(where $pos(A,z)$ is the position of the element z in the array A) transforms the specification of Q_1 into that of P_1, but does not yield a correct program. As we saw in the last chapter, to obtain a correct program we must examine the verification conditions.

Consider the path initializing the loop in P_1:

```
assert  $n \in N$
$(z,y)$ := $(A[0],0)$
suggest  $z \leq A[0:y]$, $y \in N$ .
```

We are given that $n \in N$, and must show that the loop invariants $z \leq A[0:y]$ and $y \in N$ hold after initializing z to $A[0]$ and y to 0. That is, to verify this path, we must have

$$n \in N \supset A[0] \leq A[0:0] \wedge 0 \in N .$$

Applying the transformations

$$\leq \Rightarrow \geq \text{ , } z \Rightarrow A[z] \text{ , and } 0 \Rightarrow m$$

to this condition yields

$$n \in N \supset A[m] \geq A[m:m] \wedge m \in N .$$

The first consequent $A[m] \geq A[m:m]$ is equivalent to $A[m] \geq A[m]$ and clearly holds; we are left with the consequent $m \in N$. *This* path, the, will be correct if the condition $m \in N$ is satisfied.

Next, we consider the loop-exit path

```
assert  $z \leq A[0:y]$, $y \in N$
assert  $y = n$
suggest  $z \leq A[0:n]$ ,
```

i.e. the loop invariants plus the exit test $y = n$ must imply the output specification $z \leq A[0:n]$. The transformed condition is

$$A[z] \geq A[m:y] \wedge y \in N \wedge y = n \supset A[z] \geq A[m:n] ,$$

which clearly holds.

Finally, we consider the loop-body path

```
assert $z \le A[0:y]$, $y \in N$
assert $y \ne n$
$y$ := $y+1$
if $A[y]<z$ then $z$ := $A[y]$ fi
suggest $z \le A[0:y]$, $y \in N$ .
```

To verify this path, we must show that the loop invariant continues to hold if the exit test is false and the loop body is executed, for both cases of the conditional statement. If $A[y+1]<z$ before executing the path, then the **then**-branch of the loop,

```
assert $z \le A[0:y]$, $y \in N$
assert $y \ne n$
$y$ := $y+1$
assert $A[y]<z$
$z$ := $A[y]$
suggest $z \le A[0:y]$, $y \in N$ ,
```

is taken. In that case, we must have

$$z \le A[0:y] \ \wedge \ y \in N \ \wedge \ y \ne n \ \wedge \ A[y+1]<z \ \supset \ A[y+1] \le A[0:y+1] \ \wedge \ y+1 \in N \ .$$

Similarly, for the alternative path, we need

$$z \le A[0:y] \ \wedge \ y \in N \ \wedge \ y \ne n \ \wedge \ \neg(A[y+1]<z) \ \supset \ z \le A[0:y+1] \ \wedge \ y+1 \in N \ .$$

Applying the transformations to these two conditions gives

$$A[z] \ge A[m:y] \ \wedge \ y \in N \ \wedge \ y \ne n \ \wedge \ A[y+1]<A[z] \ \supset \ A[y+1] \ge A[m:y+1] \ \wedge \ y+1 \in N$$

and

$$A[z] \ge A[m:y] \ \wedge \ y \in N \ \wedge \ y \ne n \ \wedge \ \neg(A[y+1]<A[z]) \ \supset \ A[z] \ge A[m:y+1] \ \wedge \ y+1 \in N \ .$$

Consider the second of these two verification conditions. The consequent $y+1 \in N$ clearly holds at the end of the loop body, since $y \in N$ held before the path. The consequent $A[z] \ge A[m:y+1]$ is partially implied by the conjunct $A[z] \ge A[m:y]$ appearing on the left-hand side of the implication; only $A[z] \ge A[y+1]$ is not implied. So we look for additional relations with which to complete the analogy. On the left-hand side we have the conjunct $\neg(A[y+1]<A[z])$ while the desired $A[z] \ge A[y+1]$ is equivalent to $\neg(A[y+1]>A[z])$. This suggests the additional transformation

$\langle \Rightarrow \rangle$.

With this transformation the first verification condition holds as well.

Thus the four transformations

$$(\leq \Rightarrow \geq, z \Rightarrow A[z], 0 \Rightarrow m, \langle \Rightarrow \rangle)$$

will yield a correct program for finding the position of the maximum provided that the condition $m \in N$ is satisfied. In the same manner, it may be shown that the transformations

$$(\geq \Rightarrow \leq, z \Rightarrow pos(A,z), m \Rightarrow 0, \rangle \Rightarrow \langle) \ ,$$

when applied to Q_1, will yield a correct program for finding the minimum.

Now that we have a complete analogy between the two tasks P_1 and Q_1, viz.

$$(\leq \Leftrightarrow \geq, z \Leftrightarrow A[z], 0 \Leftrightarrow m, \langle \Leftrightarrow \rangle) \ ,$$

we attempt to generalize it to obtain an abstract program schema embodying the underlying technique of the program. The generalization of the two predicates $\leq$ and $\geq$ is a new predicate variable α ; similarly, the generalization of $\langle$ and $\rangle$ is β . The generalization of z and $A[z]$ is z , since z may easily be transformed into $A[z]$ (but not vice versa); similarly, the generalization of the constant 0 and variable m is m . In this manner, we obtain the abstraction mappings

$$(\leq \Rightarrow \alpha \Leftarrow \geq, z \Rightarrow z \Leftarrow A[z], 0 \Rightarrow m \Leftarrow m, \langle \Rightarrow \beta \Leftarrow \rangle) \ ,$$

i.e.

$$(\leq \Rightarrow \alpha, 0 \Rightarrow m, \langle \Rightarrow \beta)$$

will generalize P_1 and

$$(\geq \Rightarrow \alpha, z \Rightarrow pos(A,z), \rangle \Rightarrow \beta)$$

will generalize Q_1 .

Applying the first set of transformations,

$$(\leq \Rightarrow \alpha, 0 \Rightarrow m, \langle \Rightarrow \beta) \ ,$$

to the output specification of P_1 (or the second set to Q_1) yields the abstract specification

assert $\alpha(z, A[m:n])$.

This, then, will be the output specification of the schema. To obtain the desired schema, we apply the transformations to P_1 yielding

```
$(z,y)$ := $(A[m],m)$
loop assert $\alpha(z,A[m:y])$, $y \in N$
     until $y=n$
     $y$ := $y+1$
     if $\beta(A[y],z)$ then $z$ := $A[y]$ fi
     repeat
assert $\alpha(z,A[m:n])$ .
```

Since the abstraction transformations involve constants, we must examine the schema's verification conditions. Those conditions that cannot be proved will remain as preconditions for applicability of the schema.

The verification condition for the initialization path is

$$\alpha(A[m],A[m]) \wedge m \in N \;;$$

thus, if

$$m \in N$$

and

$$\alpha(u,u)$$

for all u , then that path is correct. The condition for the exit path is

$$\alpha(z,A[m:y]) \wedge y \in N \wedge y=n \supset \alpha(z,A[m:n]) \;,$$

which is clearly the case. The remaining two conditions for the two loop-body paths are

$$\alpha(z,A[m:y]) \wedge y \in N \wedge \beta(A[y+1],z) \supset \alpha(A[y+1],A[m:y+1]) \wedge y+1 \in N$$

and

$$\alpha(z,A[m:y]) \wedge y \in N \wedge \neg\beta(A[y+1],z) \supset \alpha(z,A[m:y+1]) \wedge y+1 \in N \;.$$

The conjunct $y+1 \in N$ clearly holds in both, and since we are already requiring $\alpha(A[y+1],A[y+1])$, that leaves

$$\alpha(z,A[m:y]) \wedge y \in N \wedge \beta(A[y+1],z) \supset \alpha(A[y+1],A[m:y])$$

and

$$\alpha(z,A[m:y]) \wedge y \in N \wedge \neg\beta(A[y+1],z) \supset \alpha(A[y+1],z) \;.$$

These conditions may be generalized (cf. Boyer and Moore [1975]) by replacing the expressions $A[m:y]$ and $A[y+1]$ that appear on both sides of the implications with universally quantified variables u and v , respectively. The generalized conditions are

$$\alpha(z,u) \wedge \beta(v,z) \supset \alpha(v,u)$$

and

$$\alpha(z,u) \wedge \neg\beta(v,z) \supset \alpha(z,v)$$

(the conjuncts $y \in N$ have been dropped as y does not at all appear in the consequents). Finally, there is a termination condition

$$(\exists\zeta \in N)\ 0+\zeta=n\ ;$$

transforming it gives

$$(\exists\zeta \in N)\ m+\zeta=n\ ,$$

or equivalently, since $m \in N$,

$$n \in N \wedge m \leq n\ .$$

The schema, with its preconditions listed in its input assertion, is

```
S_1: begin comment minimum/maximum schema
    assert $\alpha(u,u)$, $\alpha(u,z) \wedge \beta(z,v) \supset \alpha(v,u)$, $\alpha(z,u) \wedge \neg\beta(v,z) \supset \alpha(z,v)$, $m,n \in N$, $m \leq n$
    $(z,y)$ := $(A[m],m)$
    loop assert $\alpha(z,A[m:y])$, $y \in N$
        until $y=n$
        $y$ := $y+1$
        if $\beta(A[y],z)$ then $z$ := $A[y]$ fi
        repeat
    assert $\alpha(z,A[m:n])$
    end .
```

Any instantiation that satisfies the preconditions is guaranteed to yield a correct program. Clearly, the predicates α and β that appear in the schema should be instantiated to primitives available in the target language, otherwise they must be replaced by equivalent predicates for the schema to yield an executable program.

In a similar manner, we could use Q_1 as the basis for the abstraction; we would obtain a somewhat different schema with the same output specification.

Note that A may be considered to be a function as any other. Thus, this schema can be instantiated to find the position or value z of the minimum or maximum of any function over the domain of integers in the range $[m:n]$. For example, to find the position z of the minimum of the function f in the interval $[0:m]$, i.e.

```
R_1: begin comment function minimum program
     assert $m \in N$
     achieve $f(z) \le f[0:m]$ varying $z$
     end ,
```

we compare this goal with the abstract specification of the schema

assert $\alpha(z, A[m:n])$.

The following instantiation is obvious:

$$(\alpha \Rightarrow \le, z \Rightarrow f(z), A \Rightarrow f, m \Rightarrow 0, n \Rightarrow m) .$$

Applying this instantiating to the preconditions

assert $\alpha(u,u)$, $\alpha(z,u) \wedge \beta(v,z) \supset \alpha(v,u)$, $\alpha(z,u) \wedge \neg\beta(v,z) \supset \alpha(z,v)$, $m, n \in N$, $m \le n$

yields

assert $u \le u$, $z \le u \wedge \beta(v,z) \supset v \le u$, $z \le u \wedge \neg\beta(v,z) \supset z \le v$, $0, m \in N$, $0 \le m$.

The first condition holds since $\le$ is reflexive; the last two follow from the input specification $m \in N$. That leaves $z \le u \wedge \beta(v,z) \supset v \le u$ and $z \le u \wedge \neg\beta(v,z) \supset z \le v$. The latter suggests completing the analogy by letting $\beta(v,z) \Rightarrow v > z$; then the other condition $z \le u \wedge z > v \supset v \le u$ holds as well.

Applying the complete instantiation mapping,

$$(\alpha \Rightarrow \le, z \Rightarrow f(z), A \Rightarrow f, m \Rightarrow 0, n \Rightarrow m, \beta \Rightarrow <) ,$$

to the schema yields

```
(f(z), y) := (f(0), 0)
loop assert $f(z) \le f[0:y]$, $y \in N$
     until $y = m$
     $y := y+1$
     if $f(y) < f(z)$ then $f(z) := f(y)$ fi
     repeat
assert $f(z) \le f[0:m]$ .
```

Replacing the illegal assignments, we get the concrete, correct program

```
R_1: begin comment function minimum program
     assert m∈N
     (z,y) := (0,0)
     loop assert f(z)≤f[0:y], y∈N
          until y=m
          y := y+1
          if f(y)<f(z) then z := y fi
          repeat
     assert f(z)≤f[0:m]
     end .
```

The same abstraction process would work were we given the two recursive programs

```
P_1'(z, A[0:n]):
begin comment recursive minimum-value program
     assert n∈N
     if n=0    then z := A[0]
               else P_1'(z, A[0:n-1])
                    if A[n]<z then z := A[n] fi
               fi
     assert z≤A[0:n], n∈N
     end
```

and

```
Q_1'(z, A[m:n]):
begin comment recursive maximum-position program
     assert m,n∈Z, m≤n
     if m=n    then z := n
               else Q_1'(z, A[m+1:n])
                    if A[m]>A[z] then z := m fi
               fi
     assert A[z]≥A[m:n], m∈N
     end .
```

Abstracting these two programs, we would obtain the scheme

```
$S_1'(z, A[m:n])$:
begin comment *recursive minimum/maximum schema*
    assert $\alpha(u,u)$, $\alpha(z,u) \wedge \beta(v,z) \supset \alpha(v,u)$, $\alpha(z,u) \wedge \neg\beta(v,z) \supset \alpha(z,v)$, $m, n \in Z$, $m \le n$
    if $n=m$    then $z := A[m]$
                else $P_1'(z, A[m:n-1])$
                     if $\beta(A[n], z)$ then $z := A[n]$ fi
                fi
    assert $\alpha(z, A[m:n])$, $n \in N$
    end .
```

Example 2: *Associative Recursicr. Schema.*

Consider the two programs:

```
$P_2$: begin comment *factorial program*
    assert $a \in N$
    $(z,y) := (1,a)$
    loop assert $y! \cdot z = a!$
         until $y=0$
         $(z,y) := (y \cdot z, y-1)$
         repeat
    assert $z=a!$
    end
```

and

```
Q₆: begin comment array-summation program
    assert $n \in \mathbb{N}$
    $(z,y) := (0,m)$
    loop assert $\Sigma_{\zeta=y}^{n} A[\zeta]+z=\Sigma_{\zeta=m}^{n} A[\zeta]$
        until $y=n+1$
        $(z,y) := (A[y]+z, y+1)$
        repeat
    assert $z=\Sigma_{\zeta=m}^{n} A[\zeta]$
    end .
```

Matching the two output specifications

assert $z=a!$

and

assert $z=\Sigma_{\zeta=m}^{n} A[\zeta]$

suggests, as one possible analogy,

$$(u! \Leftrightarrow \Sigma_{\zeta=u}^{n} A[\zeta], a \Leftrightarrow m) .$$

The two functions $u!$ and $\Sigma_{\zeta=u}^{n} A[\zeta]$ generalize to a function variable $f(u)$; the input variables a and m generalize to, say, a :

$$(u! \Rightarrow f(u) \Leftarrow \Sigma_{\zeta=u}^{n} A[\zeta], a \Rightarrow a \Leftarrow m) .$$

Thus, we get the abstract output specification

assert $z=f(a)$.

Both programs consist of a single loop; their respective loop invariants are

assert $y!\cdot z=a!$

and

assert $\Sigma_{\zeta=y}^{n} A[\zeta]+z=\Sigma_{\zeta=m}^{n} A[\zeta]$.

Matching the invariants, after applying the transformations already found, gives

$$f(y)\cdot z=f(a) \Leftrightarrow f(y)+z=f(a) ,$$

and we derive the additional aspect of the analogy

$$\cdot \Rightarrow h \Leftarrow + \ .$$

Applying the corresponding transformations to the loop invariants we obtain the abstract invariant

$$\textbf{assert}\ \ h(f(y),z)=f(a)\ .$$

Now, we must consider the verification conditions. The initialization condition of P_2 is

$$a!\cdot 1=a!\ ,$$

and applying the transformations we get

$$h(f(a),1)=f(a)\ ;$$

on the other hand, applying the transformations to the initialization condition of Q_2,

$$\Sigma_{\zeta=m}^{n}A[\zeta]+0=\Sigma_{\zeta=m}^{n}A[\zeta]\ ,$$

gives

$$h(f(a),0)=f(a)\ .$$

To unify the two, we add to the analogy

$$1 \Rightarrow e \Leftarrow 0\ ,$$

and obtain the abstract condition

$$h(f(a),e)=f(a)\ .$$

The loop-exit condition derived from P_2 is

$$h(f(y),z)=f(a)\ \wedge\ y=0\ \supset\ z=f(a)\ ;$$

from Q_2, we get

$$h(f(y),z)=f(a)\ \wedge\ y=n+1\ \supset\ z=f(a)\ .$$

With the abstraction

$$0 \Rightarrow n \Leftarrow n+1\ ,$$

we get

$$h(f(y),z)=f(a)\ \wedge\ y=n\ \supset\ z=f(a)\ ,$$

or, equivalently

$$h(f(n),z)=z\ .$$

For the loop-body paths, we have the conditions

$$h(f(y),z)=f(a) \ \wedge\ y\neq n \ \supset\ h(f(y-1),h(y,z))=f(a)$$

and

$$h(f(y),z)=f(a) \ \wedge\ y\neq n \ \supset\ h(f(y+1),h(A[y],z))=f(a) \ ,$$

for P_2 and Q_2, respectively. To unify $y-1 \Leftrightarrow y+1$ and $y \Leftrightarrow A[y]$, we need the additional abstractions

$$(+ \Rightarrow g \Leftarrow -, \varphi \Rightarrow i \Leftarrow A) \ ,$$

where φ is the identity function, i.e. $\varphi(u)=u$. This yields

$$h(f(y),z)=f(a) \ \wedge\ y\neq n \ \supset\ h(f(g(y,1)),h(i(y),z))=f(a) \ ,$$

or equivalently

$$y\neq n \ \supset\ h(f(g(y,1)),h(i(y),z))=h(f(y),z) \ .$$

The complete abstraction is

$$(u! \Rightarrow f(u) \Leftarrow \Sigma_{\zeta=u}^{n}A[\zeta],\, a \Rightarrow a \Leftarrow m,\, \cdot \Rightarrow h \Leftarrow +,\, 1 \Rightarrow e \Leftarrow 0,$$
$$0 \Rightarrow n \Leftarrow n+1,\, + \Rightarrow g \Leftarrow -,\, \varphi \Rightarrow i \Leftarrow A) \ .$$

Applying it to Q_2, and collecting all the preconditions, we derive the schema

```
S_2: begin comment associative recursion schema
       assert $h(u,e)=u,\ h(f(n),z)=z,\ y\neq n \supset h(f(g(y,1)),h(i(y),z))=h(f(y),z)$
       $(z,y)$ := $(e,a)$
       loop assert $h(f(y),z)=f(a)$
            until $y=n$
            $(z,y)$ := $(h(i(y),z),g(y,1))$
            repeat
       assert $z=f(a)$
       end .
```

In this manner we have obtained a general schema for computing a function $f(a)$. It applies to recursive functions $f(x)$ such that $f(n)=e$ is a unit of an associative and commutative function h, and $f(u)=h(f(g(u,1)),i(u))$ when $y\neq n$. The schema is similar to one of the recursion-to-iteration transformations of Burstall and Darlington [1977].

To see how this schema may applied to another problem, consider the specifications

```
R_2: begin comment list-reversal program
       assert r∈𝓛
       achieve z=reverse(r) varying z
       end ,
```

where $\mathcal{L}$ is the set of all lists, and $reverse(r)$ is a list containing the elements of r in reverse order. Assume that we are also given two relevant facts about $reverse$:

fact $reverse(()) = ()$

and

fact $reverse(u) = reverse(tail(u)) \cdot (head(u))$ **when** $u \neq ()$,

where $u \cdot v$ concatenates the two lists u and v , $head(u)$ is the first element of the list u , and $tail(u)$ is a list of all but the first element, and $()$ is the empty list.

An initial comparison of the schema's output specification $z=f(a)$ with the new specification $z=reverse(r)$ suggests the instantiation

$$f \Rightarrow reverse \ .$$

Instantiating the precondition

$$y \neq n \supset h(f(g(y,1)), h(i(y),z)) = h(f(y),z)$$

gives

$$y \neq n \supset h(reverse(g(y,1)), h(i(y),z)) = h(reverse(y),z) \ .$$

By the second of the above two facts, we have that $reverse(y)$ may be replaced by $reverse(tail(y)) \cdot (head(y))$, provided that y is not the empty list $()$. This suggests instantiating $n \Rightarrow ()$ to obtain

$$y \neq () \supset h(reverse(g(y,1)), h(i(y),z)) = h(reverse(tail(y)) \cdot (head(y)), z) \ .$$

The function $reverse$ appears on the two sides of the equality, so we try to generalize this condition by replacing both occurrences of $reverse$ with an arbitrary list u . To do that, we must first unify $reverse(g(y,1))$ with $reverse(tail(y))$ by instantiating $g(u,v) \Rightarrow tail(u)$. We are left with the condition

$$h(u, h(i(y),z)) = h(u \cdot (head(y)), z) \ .$$

Similarly, we unify $i(u)$ with $(head(u))$, the list containing just the first element of u , obtaining

$$h(u, h(v,z)) = h(u \cdot v, z) \ .$$

This matches with

fact $u \cdot (v \cdot w) = (u \cdot v) \cdot w$,

by instantiating $h \Rightarrow \cdot$, i.e. $\cdot$ is associative.

The instantiations we have found are

$$(f \Rightarrow reverse, n \Rightarrow (), g(u,v) \Rightarrow tail(u), i(u) \Rightarrow (head(u)), h \Rightarrow \cdot) .$$

Applying them to the other preconditions

assert $h(u,e)=u,\ \ h(f(n),z)=z$

yields

assert $u \cdot e = u,\ \ reverse(()) \cdot z = z$.

But $reverse(())=()$ and $() \cdot z = z$, since the empty list $()$ is an identity element of the function   Thus, the second condition holds; the first suggests letting $e \Rightarrow ()$.

The completed instantiation is

$$(f \Rightarrow reverse, n \Rightarrow (), g(u,v) \Rightarrow tail(u), i(u) \Rightarrow (head(u)), h \Rightarrow \cdot, e \Rightarrow ()) .$$

In all, we have derived the following program

```
R2: begin comment list reversal program
    assert r∈ℒ
    z := ()
    y := r
    loop assert reverse(y)•z=reverse(r)
         until y=()
         z := (head(y))•z
         y := tail(y)
         repeat
    assert z=reverse(r)
    end .
```

Example 3: *Binary-Search Schema.*

In the general overview, we saw how a binary-search schema was abstracted from two programs, one for real division and the other for square roots. In this example, we shall begin with the array-search program instead of the square-root one; a more general schema will result.

Consider the following two annotated binary-search programs, P_5 and Q_5 :

```
P5: begin  comment real-division program
    assert  $0 \le a < b,\ 0 < e$
    $(z, y) := (0, 1)$
    loop assert  $b \cdot z \le a,\ a < b \cdot (z+y)$
        until  $y \le e$
        $y := y/2$
        if  $b \cdot (z+y) \le a$  then  $z := z+y$  fi
        repeat
    assert  $b \cdot z \le a,\ a < b \cdot (z+e)$
    end
```

and

```
Q5: begin  comment array-search program
    assert  $u \le v \supset A[u] \le A[v],\ A[u] \in \mathbb{N},\ b \in bag(A[1{:}n])$
    $(z, y) := (n, n)$
    loop assert  $z \ge pos(A, b),\ pos(A, b) > z-y$
        until  $y \le 1$
        $y := y/2$
        if  $A[z-y+1] > b$  then  $z := z-y$  fi
        repeat
    assert  $A[z] \le b,\ b < A[z+1]$
    end .
```

Recall that when an index u of an array A is not an integer, the intended element is $A\lfloor u \rfloor$.

The analogy between the specification of P_5 ,

assert $b \cdot z \le a,\ a < b \cdot (z+e)$

and the specification of Q_5 ,

assert $A[z] \le b,\ b < A[z+1]$,

is

$$(u \cdot v \le w \Leftrightarrow w[v] \le u,\ a \Leftrightarrow A,\ u < v \cdot w \Leftrightarrow v < u[w],\ e \Leftrightarrow 1) .$$

The corresponding abstraction mappings are

$$(u \cdot v \le w \Rightarrow \alpha(u,v,w) \Leftarrow w[v] \le u,\ a \Rightarrow a \Leftarrow A,$$
$$u < v \cdot w \Rightarrow \beta(u,v,w) \Leftarrow v < u[w],\ e \Rightarrow e \Leftarrow 1) \ .$$

Applying these transformations to P_3 yields the schema

```
(z,y) := (0,1)
loop assert $\alpha(b,z,a)$, $\beta(a,b,z+y)$
     until $y \le e$
     y := y/2
     if $\alpha(b,z+y,a)$ then z := z+y fi
     repeat
assert $\alpha(b,z,a)$, $\beta(a,b,z+e)$ .
```

In the same manner as in previous examples, we derive the precondition

$$\beta(a,b,u) \wedge u \le v \supset \beta(a,b,v)$$

for the loop-exit path, and

$$\neg\alpha(b,u,a) \supset \beta(a,b,u)$$

for the loop-body path.

The verification condition for the loop-initialization path is

$$\alpha(b,0,a) \wedge \beta(a,b,1) \ .$$

However, if we were to abstract Q_3 instead, we would get an initialization condition

$$\alpha(b,n,a) \wedge \beta(a,b,0) \ .$$

This suggests generalizing the constant 0 in P_3 and n in Q_3 to j and 1 and 0 to k. In this manner, we would obtain the initialization

$$(z,y) := (j,k)$$

with the unified preconditions

$$\alpha(b,j,a) \wedge \beta(a,b,k) \ .$$

Alternatively, we can just preface the loop with an unachieved subgoal

achieve $\alpha(b,z,a),\ \beta(a,b,z+y)$ **varying** z,y ,

stating that the loop invariant must be achieved before entering the loop.

Adopting the second option we get the schema

```
S_5: begin comment binary-search schema
    assert $\beta(a,b,u) \wedge u \le v \supset \beta(a,b,v)$,  $\neg\alpha(b,u,a) \supset \beta(a,b,u)$
    achieve $\alpha(b,z,a)$,  $\beta(a,b,z+y)$ varying $z, y$
    loop assert $\alpha(b,z,a)$,  $\beta(a,b,z+y)$
         until $y \le e$
         $y$ := $y/2$
         if $\alpha(b,z+y,a)$ then $z$ := $z+y$ fi
         repeat
    assert $\alpha(b,z,a)$,  $\beta(a,b,z+e)$
    end .
```

It is a general program schema for a binary search within a tolerance with the abstract output specification

assert $\alpha(b,z,a)$, $\beta(a,b,z+e)$.

To illustrate how this search schema may be used, we consider a variation on the square-root program:

```
R_5: begin comment variant square-root program
    assert $0<d$,  $1<c$
    achieve $z-d<\sqrt{c}$,  $\sqrt{c} \le z$ varying $z$
    end ,
```

that is, the result z may only be greater than the square-root of c by less than the given d . We would like to instantiate the binary-search schema to yield such a square-root program.

In order to match this output specification with that of our schema:

assert $\alpha(b,z,a)$, $\beta(a,b,z+e)$,

we let the constant e be the constant expression $-d$ and obtain the transformations:

$(a \Rightarrow c, \alpha(b,z,u) \Rightarrow \sqrt{u} \le z, \beta(u,b,v) \Rightarrow v<\sqrt{u}, e \Rightarrow -d)$.

The preconditions

assert $\beta(a,b,u) \wedge u \le v \supset \beta(a,b,v)$, $\neg\alpha(b,u,a) \supset \beta(a,b,u)$

instantiate to

assert $u<\sqrt{c} \wedge u \le v \supset v<\sqrt{c}$, $\neg(\sqrt{c} \le u) \supset u<\sqrt{c}$.

The second condition holds, while the first does not. To get the first condition to hold, we need $u \geq v$, rather than $u \leq v$, suggesting the additional transformation $\leq \Rightarrow \geq$.

To satisfy the initialization condition, we need to

achieve $\alpha(b, z, a)$, $\beta(a, b, z+y)$ **varying** z, y ,

i.e.

achieve $\sqrt{c} \leq z$, $z+y < \sqrt{c}$ **varying** z, y .

We note that since $1 < c$, we have $\sqrt{c} < c$ and $c+(1-c)=1<\sqrt{c}$. Thus, both conjuncts hold when we let:

(z, y) := $(c, 1-c)$.

(An alternative would have been to take $-c$ for y , since $c+(-c)=0<\sqrt{c}$.)

The instantiated schema is:

```
assert $0<d$, $1<c$
$(z,y)$ := $(c, 1-c)$
loop assert $\sqrt{c} \leq z$, $z+y<\sqrt{c}$
     until $y \geq -d$
     $y$ := $y/2$
     if $\sqrt{c} \leq z+y$ then $z$ := $z+y$ fi
     repeat
assert $\sqrt{c} \leq z$, $z-d<\sqrt{c}$ .
```

However, since α involves the square-root function itself, the conditional test is not primitive and must be replaced. *It can be replaced by $c \leq (z+y)^2$ since c and $z+y$ are* nonnegative ($0 \leq c$ follows from the input specification; the relation $0 \leq z+y$ may be shown to be a global invariant). Thus, we have:

```
R₃: begin comment variant square-root program
    assert $0<d$, $1<c$
    $(z,y)$ := $(c, 1-c)$
    loop assert $\sqrt{c}\le z$, $z+y<\sqrt{c}$
         until $y\ge -d$
         $y$ := $y/2$
         if $c\le(z+y)^2$ then $z$ := $z+y$ fi
         repeat
    assert $\sqrt{c}\le z$, $z-d<\sqrt{c}$
    end .
```

In the last two chapters, we have explored modification and abstraction techniques; in the next two chapters we develop helpful tools for synthesizing and annotating programs.

CHAPTER V

PROGRAM SYNTHESIS

1. INTRODUCTION

Recently, researchers have tried to gain insight into the haphazard art of programming. This has led to the development of "structured programming" which has been defined by Hoare as "the task of organizing one's thought in a way that leads, in a reasonable time, to an understandable expression of a computing task". One of the guidelines of structured programming is that "one should try to develop a program and its proof of correctness hand-in-hand" (Gries [1974]). Much has been written on the subject, including the works of Dijkstra [1968,1976], Dahl, Dijkstra, and Hoare [1972], Wirth [1973,1974], Conway and Gries [1973], and others.

The idea is to construct the desired program step by step, beginning with the given input and output specifications. In each step the current goal is solved, transformed into another goal, or reduced to simpler subgoals. Each stage is correct if its predecessor is, thereby guaranteeing the correctness of the final program. Our purpose in this chapter is to formalize some of the strategies of structured programming, thereby contributing to its automation. As we have seen, such methods are needed to complement the techniques of program modification and instantiation.

One of the major hurdles in automatic structured programming lies in the formation of loops. Recent synthesis systems have variously dealt with this problem. Buchanan and Luckham [1974] require the user to supply the skeleton of the loop, and the system fills in the details. Sussman [1975] described his HACKER system that creates iterative and recursive loops with no guarantee of correctness. Darlington [1975], Manna and Waldinger [1975,1977], and others have described a technique of recursion formation and the need to sometimes strengthen the original specifications for that purpose. The system described in Green [1976] assumes extensive a priori programming knowledge, such as an experienced programmer would have. Duran [1975] investigated the use of loop invariants in the synthesis of programs, along lines similar to our iterative loop strategy. For a survey of these and other approaches to automatic program synthesis, see Biermann [1976].

The next section contains an overview of the steps involved in the synthesis of a simple program. In Section 3, we introduce some programming rules; in the fourth section, the rules are employed in the syntheses of several programs. A final section deals with the problem of extending a program to achieve additional goals. When synthesizing code to extend a program, care must be taken to ensure that the original specifications continue to be satisfied.

2. OVERVIEW

In this overview, we informally describe the synthesis of a simple program; the steps in the strategies themselves are explained in the next section.

Program synthesis begins with an initial goal of the form

```
P: begin comment desired program
     assert input specification
     achieve output specification varying output variables
     end ;
```

The task is to expand the goal into a segment of code whose execution will terminate with the relation expressed in the output specification holding between the variables. By

varying *output variables* ,

we indicate that only those variables may be set by the program; other variables appearing in the specifications are input variables. The statement

assert *input specification*

specifies the set of values of the input variables for which the synthesized program is expected to work.

From these specifications, an annotated program of the form

```
assert input specification
purpose output specification
     code to achieve specifications
assert output specification
```

is constructed. When control reaches the end of the program, the variables must satisfy the output specification. The code must be primitive, in other words, it may not itself contain **achieve** statements or nonprimitive operators. Thus, a program synthesizer "compiles" the high-level **achieve** statements into lower-level "code". The **purpose** statement is a comment expressing what it is that the following code was intended to achieve.

It is usually not possible to generate code directly from the initial goal. Rather, at each stage of the construction, a current goal is replaced by one or more new, and hopefully more readily achievable, subgoals, that if and when achieved will imply the desired relation. Each step must preserve correctness, i.e. satisfying the new goals must yield a correct program satisfying the current goal. Thus, the final program is guaranteed to satisfy the original specifications.

In general, at each stage in the synthesis of a program there is more than one unachieved subgoal, and for each subgoal there may be a number of possible transformations that can be applied. Whenever a given choice turns out to be unsuccessful, a different possibility must be tried. We do not, however, address here the important issue of how to guess which may be the best choice at any particular point. Kant [1977] describes a system that guides the choices made by a synthesis system based upon an analysis of expected time and space requirements. Annotation techniques facilitate such analyses and could be employed in conjunction with the synthesis.

Consider the goal

```
P_0: begin comment gcd program
     assert a∈N, b∈N+1
     achieve z=gcd(a,b) varying z
     end
```

(where N+1 is the set of positive integers), aimed at constructing a program that sets the variable z to the greatest common divisor (gcd) of two nonnegative integers a and b.

Were gcd a primitive function of the target language, then this goal could be achieved by a simple assignment statement:

$$z := gcd(a,b) .$$

But having no primitive gcd function available, the goal must be achieved in stages utilizing domain-specific knowledge about gcd. Furthermore, we assume that the set constructor $\{\ldots\}$ and max function are not primitive, or else we could use the definition

$$\textbf{fact}\ \ gcd(u,v)=max\{w\in N:w|u\wedge w|v\}\ \ \textbf{when}\ \ u,v\in Z$$

(where the predicate $w|u$ means that w divides u evenly) to assign

$$z := max\{w\in N:w|a\wedge w|b\} .$$

Note also, that were it not specified that only z may be varied, the goal could be achieved by the assignments

$$(z,a) := (b,0)$$

since $a=0$ and $z=b$ imply $z=gcd(a,b)$.

Not having any way to directly

achieve $z=gcd(a,b)$ **varying** z ,

the first step in the synthesis might be to introduce program variables whose values may be manipulated by the program so as to achieve the goal. To this end, the goal may be replaced by the conjunction of two subgoals

achieve $z=gcd(s,t)$, $u=gcd(s,t) \supset u=gcd(a,b)$ **varying** z,s,t .

The second subgoal $u=gcd(s,t) \supset u=gcd(a,b)$ requires that for any value u , if u is equal to $gcd(s,t)$ — for some values of the new program variables s and t — then it is also equal to the desired value $gcd(a,b)$. In particular, if the variable z has the value $gcd(s,t)$ at the same time as the second subgoal holds, then the original goal $z=gcd(a,b)$ is satisfied. Since the implication $u=gcd(s,t) \supset u=gcd(a,b)$ must hold for all values of u , it is equivalent to the simpler $gcd(s,t)=gcd(a,b)$. Our current goal, then, is

achieve $z=gcd(s,t)$, $gcd(s,t)=gcd(a,b)$ **varying** z,s,t .

At this point, we would like to simplify this goal, which is composed of two conjuncts, by splitting it into two consecutive nonconjunctive subgoals. Choosing to first achieve $gcd(s,t)=gcd(a,b)$ and then $z=gcd(s,t)$, we get the two subgoals

achieve $gcd(s,t)=gcd(a,b)$ **varying** s,t
achieve $z=gcd(s,t)$ **varying** z,s,t ,

each of which is simpler than the conjunctive goal. However, in so doing, one must ensure that achieving the second subgoal will not "clobber" what was accomplished by the first subgoal. This point will be taken up later.

To achieve the first subgoal

achieve $gcd(s,t)=gcd(a,b)$ **varying** s,t ,

it suffices to

achieve $(s,t)=(a,b)$ **varying** s,t ;

to achieve the latter, we can assign

(s,t) := (a,b) .

Matching $z=gcd(s,t)$ with the domain-specific knowledge

fact $gcd(0,u)=u$ **when** $u \in N+1$,

stating that if u is positive, then the gcd of u and 0 is u , we get $(s \Rightarrow 0, t \Rightarrow u, z \Rightarrow u)$, i.e. $z=gcd(s,t)$ if $s=0$ and $z=t \in N+1$. *Thus*, the remaining subgoal

achieve $z=gcd(s,t)$ **varying** z,s,t

may be replaced by the sufficient

achieve $z=t,\ t\in N+1,\ s=0$ **varying** z,s,t .

Again, we may split the conjunctive goal into two consecutive subgoals

achieve $s=0,\ t\in N+1$ **varying** s,t
achieve $z=t$ **varying** z .

Notice that we have allowed the first subgoal to vary s and t, but not z, while the second goal may vary only z, leaving the values of s and t unchanged. Achieving the second goal, say by assigning

$z := t$,

will therefore leave $s=0$ and $t\in N+1$ once those relations have been achieved by the preceding subgoal.

It remains to

achieve $s=0,\ t\in N+1$ **varying** s,t .

To achieve $s=0$, we do not want to simply assign $s:=0$, since this will undo the previous assignment $s:=a$; on the other hand, we must vary the value of s since a is not necessarily 0 . To resolve this dilemma, recall that we set $(s,t)=(a,b)$ only in order to achieve the relation $gcd(s,t)=gcd(a,b)$. So, if we can "protect" this latter relation while achieving $s=0$, rather than protect the stronger $(s,t)=(a,b)$, then when $s=0$ is achieved, the desired relation $gcd(s,t)=gcd(a,b)$ will still hold. The protected relation $gcd(s,t)=gcd(a,b)$ is termed an *invariant assertion*; it is associated with a specific point in the program segment, and expresses that part of the goal that has already been computed whenever execution reaches that point. (It is the *inductive assertion* used in Floyd's [1967] method of proving program correctness.)

An alternative, but equivalent, way of viewing this solution is as follows: The original purpose in having introduced the variables s and t and set $(s,t)=(a,b)$ was to enable us to compute $z=gcd(s,t)$ rather than $z=gcd(a,b)$. So, we must make sure that, though the value of s is changed, it still suffices to achieve $z=gcd(s,t)$, i.e., achieving $z=gcd(s,t)$ for the new value of s will imply $z=gcd(s,t)$ for the old value as well. The relation $z=gcd(s,t)$ is then called an *invariant purpose*; it expresses the ultimate goal throughout the computation. (It is the assertion used for *subgoal induction*, see Manna [1971] and Morris and Wegbreit [1977]).

To achieve $s=0$ while protecting the invariant relations, we construct a loop of the form

```
loop assert $gcd(s,t)=gcd(a,b)$
     purpose $z=gcd(s,t)$
     until $s=0$
     approach $s=0$ varying $s,t$
     repeat .
```

The statement

assert $gcd(s,t)=gcd(a,b)$

contains the invariant assertion of the loop; the statement

purpose $z=gcd(s,t)$

contains the invariant purpose. For a relation to be an invariant assertion of a loop, it must hold upon entering the loop, and assuming that it held before executing the loop body, then it must hold after. For a relation to be an invariant purpose of a loop, it must be the goal upon entering the loop, and assuming that it is the goal before executing the loop body, then it must be the goal after.

The loop-body statement

approach $s=0$ **varying** s,t

expresses the desire to make definite progress towards the goal $s=0$ with each loop iteration. Since s is set to the nonnegative integer a before entering the loop, it follows that the loop decreases the value of s. Thus, we can ensure loop termination by monotonically decreasing the integer s, while s remains nonnegative; the loop-body subgoal, then, is

achieve $s\in\mathbf{N}$, $s<s'$ **varying** s,t ,

where s' denotes the value of s prior to this statement.

The other conjunct of the subgoal $t\in\mathbf{N}+1$ is true upon entering the loop, when $t=b\in\mathbf{N}+1$, and must be kept true by the loop. Within the loop, we also wish to protect the invariant assertion $gcd(s,t)=gcd(a,b)$ and invariant purpose $z=gcd(s,t)$. If $gcd(s',t')=gcd(a,b)$ holds for the prior values of s and t , then $gcd(s,t)=gcd(a,b)$ will hold for the new values, provided that $gcd(s,t)=gcd(s',t')$. This relation also maintains the goal $z=gcd(s,t)$: if $gcd(s,t)=gcd(s',t')$ and the goal $z=gcd(s,t)$ can be achieved, then $z=gcd(s',t')$ will be achieved as well. The complete loop-body subgoal is

achieve $s\in\mathbf{N}$, $s<s'$, $t\in\mathbf{N}+1$, $gcd(s,t)=gcd(s',t')$ **varying** s,t .

Matching the information about the domain expressed in the

fact $gcd(rem(u,v),v)=gcd(v,u)$ **when** $u\in\mathbf{N}$, $v\in\mathbf{N}+1$

with the conjunct $gcd(s,t)=gcd(s',t')$ suggests letting $t=s'$ and $s=rem(t',s')$, leaving the goal

achieve $s \in N,\ s<s',\ t \in N+1,\ t=s',\ s=rem(t',s'),\ t' \in N$ **varying** s,t .

Since $s=rem(t',s')$ and

fact $rem(u,v)<v,\ rem(u,v) \in N$ **when** $u \in N,\ v \in N+1$,

the conjuncts $s \in N$ and $s<s'$ hold, provided that $t' \in N$ and $s' \in N+1$. Recall that we are assuming that the invariants $t' \in N+1$ and $s' \in N$ are true. For the loop-body to be executed, the exit test $s=0$ must have been false, i.e. $s' \neq 0$; therefore, $s' \in N+1$. Finally, we are left with the goal

achieve $t=s',\ s=rem(s',t')$ **varying** s,t ,

suggesting the multiple assignment

(s,t) := $(rem(s,t),s)$.

Since for each step in the construction, achieving the new subgoals satisfies the previous goal, the final program,

```
P0: begin comment gcd program
    assert a,b∈N, a≠0∨b≠0
    purpose z=gcd(a,b)
        purpose gcd(s,t)=gcd(a,b)
            (s,t) := (a,b)
        assert gcd(s,t)=gcd(a,b)
        loop assert gcd(s,t)=gcd(a,b)
            purpose z=gcd(s,t)
            until s=0
            (s,t) := (rem(s,t),s)
            repeat
        z := t
    assert z=gcd(a,b)
    end ,
```

is guaranteed to achieve the initial specifications

assert $a,b \in N$
achieve $z=gcd(a,b)$ **varying** z .

In the next section, we formalize our program-synthesis strategies.

3. STRATEGIES

In this section we present some programming strategies; each transforms a given goal into code containing simpler subgoals.

1. *Strengthening Rule*

The *strengthening rule* is used to replace a goal by another sufficient goal:

$$\frac{\begin{array}{l}\textbf{achieve}\ \alpha(\bar{u})\ \textbf{varying}\ \bar{u}\\[1ex] \textbf{fact}\ \alpha(\bar{u})\ \textbf{when}\ \beta(\bar{u})\end{array}}{\begin{array}{l}\textbf{purpose}\ \alpha(\bar{u})\\ \quad\quad \textbf{achieve}\ \beta(\bar{u})\ \textbf{varying}\ \bar{u}\\ \textbf{assert}\ \alpha(\bar{u})\ ,\end{array}}$$

This rule states that if it is known that achieving β will imply that the desired relation α holds, then replace the goal

achieve $\alpha(\bar{u})$ **varying** $\bar{u}$

with the "stronger", but presumably simpler, subgoal

achieve $\beta(\bar{u})$ **varying** $\bar{u}$.

For example, the goal

achieve $z{=}gcd(x,y)$ **varying** x,y,z

may be strengthened to

achieve $x{=}0,\ z{=}y$ **varying** x,y,z ,

since the

fact $gcd(0,u){=}u$

tells us that $z{=}gcd(x,y)$ when $x{=}0$ and $z{=}y$.

The goal β may also introduce new variables $\bar{v}$, yielding

achieve $\beta(\bar{u},\bar{v})$ **varying** $\bar{u},\bar{v}$.

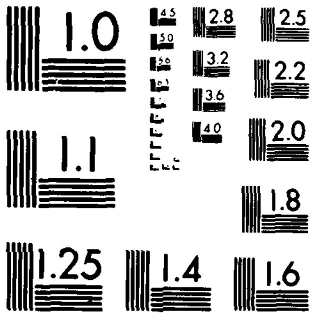
The values of the new program variables are set by the code generated from this subgoal, but β must imply α for *any* values of $\bar{v}$. Note that this means that β need only be

AD-A082 359 WEIZMANN INST OF SCIENCE REHOVOTH (ISRAEL) DEPT OF --ETC F/G 9/2
SYNTHESIZED STRUCTURED PROGRAMMING.(U)
JAN 80 Z MANNA AFOSR-78-3483

UNCLASSIFIED RADC-TR-79-326 NL

2 of 2
AD
A082359

END
DATE
FILMED
4 80
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

achieved for *some* values of $\bar{v}$; their final values are unimportant as they are not output variables. Some generally useful transformations of this sort are expressed by the following facts:

> **fact** $p(\bar{u})$ **when** $p(\bar{v}),\ p(\bar{v}) \supset p(\bar{u})$,
> **fact** $p(\bar{u})$ **when** $p(\bar{v}),\ \bar{u}=\bar{v}$,
> **fact** $p(f(\bar{u}))$ **when** $p(f(\bar{v})),\ f(\bar{u})=f(\bar{v})$,

and

> **fact** $p(f(\bar{u}))$ **when** $p(v),\ v=f(\bar{u})$.

For example, the goal

> **achieve** $z=gcd(a,b)$ **varying** z

may be transformed into

> **achieve** $z=gcd(s,t),\ (s,t)=(a,b)$ **varying** z,s,t .

or to

> **achieve** $z=gcd(s,t),\ gcd(s,t)=gcd(a,b)$ **varying** z,s,t

In any case, $z=gcd(a,b)$ is implied for any values of s and t.

A special case of this rule is the replacement of a goal by a logically equivalent, but simpler, goal. For example

> **achieve** $x=0,\ x=y$ **varying** x,y

may be replaced by the equivalent

> **achieve** $x=0,\ y=0$ **varying** x,y .

2. *Assignment Rule*

Assignment statements are formed by the following rule

$$\frac{\textbf{achieve}\ y_1=f_1(\bar{x}),\ y_2=f_2(\bar{x}),\ \ldots,\ y_n=f_n(\bar{x})\ \textbf{varying}\ y_1,y_2,\ \ldots,y_n,\ \ldots}{\begin{array}{l}\textbf{purpose}\ y_1=f_1(\bar{x}),\ y_2=f_2(\bar{x}),\ \ldots,\ y_n=f_n(\bar{x}) \\ \quad (y_1,y_2,\ \ldots,y_n) := (f_1(\bar{x}),f_2(\bar{x}),\ \ldots,f_n(\bar{x})) \\ \textbf{assert}\ y_1=f_1(\bar{x}),\ y_2=f_2(\bar{x}),\ \ldots,\ y_n=f_n(\bar{x})\ ,\end{array}}$$

where the variables $y_1, y_2, \ldots, y_n$ do not appear in $\bar{x}$, and $f_1, f_2, \ldots, f_n$ are composed of only primitive operations. For example, the goal

```
    achieve $x=0$, $y=0$ varying $x,y$
```

may be achieved by

```
    $(x,y)$ := $(0,0)$ .
```

This rule suggests that one first attempt to isolate variables on one side of an equality, e.g. a goal of the form

```
    achieve $g(y)=f(x)$ varying $y$
```

should be transformed into

```
    achieve $y=g^{-}(f(x))$ varying $y$ ,
```

where g^{-} is the inverse (assuming that it exists) of the function g.

3. *Conditional Rule*

Conditional statements are formed in the following manner:

```
purpose $\alpha(\bar{u})$
    achieve $\beta(\bar{u})$, $\gamma(\bar{u})$ varying $\bar{u}$
-------------------------------------------------------------------
purpose $\alpha(\bar{u})$
    if $\beta(\bar{u})$ then achieve $\gamma(\bar{u})$ protecting $\beta(\bar{u})$ varying $\bar{u}$
                         else assert $\neg\beta(\bar{u})$
                              achieve $\alpha(\bar{u})$ varying $\bar{u}$
                              fi ,
```

provided that the relation β is computable, i.e. when β is composed of primitive functions and predicates. In other words, one way of achieving β is to test if it holds: when it does, protect that relation while achieving the remainder; when it does not hold, try to use that fact while achieving the original goal.

For example, to solve the conjunctive goal

```
    purpose $z=gcd(x,y)$
        achieve $x=0$, $z=y$ varying $z$ ,
```

we may test if one conjunct already holds:

```
purpose $z=gcd(x,y)$
    if $x=0$   then  achieve $z=y$ varying $z$
               else  assert $x\neq 0$
                     achieve $z=gcd(x,y)$ varying $z$
               fi .
```

4. *Splitting Strategies*

Suppose we have a conjunctive goal of the form

achieve $\beta(\bar{v})$, $\gamma(\bar{u},\bar{v})$ **varying** $\bar{u},\bar{v}$,

the $\bar{u}$ variables appearing only in γ. We would like to split this goal into two consecutive subgoals, first achieving β and then γ :

```
purpose $\beta(\bar{v})$, $\gamma(\bar{u},\bar{v})$
    achieve $\beta(\bar{v})$ varying $\bar{v}$
    achieve $\gamma(\bar{u},\bar{v})$ varying $\bar{u},\bar{v}$
assert $\beta(\bar{v})$, $\gamma(\bar{u},\bar{v})$ .
```

Unfortunately, things are not as simple as that. As pointed out earlier, the problem is that in achieving the second goal γ , we may unwittingly destroy the relationship β that has already been achieved. We must, therefore, somehow maintain β while achieving γ . We consider three "protection" strategies for achieving the second subgoal, γ , while protecting the first, β , from being undone.

- **Disjoint Goal Rule.** If in achieving the second subgoal γ, the value of $\bar{v}$ need not be set, then clearly the two subgoals are independent. We have then the consecutive goals

```
achieve $\beta(\bar{v})$, $\gamma(\bar{u},\bar{v})$ varying $\bar{u},\bar{v}$
-----------------------------------------------
purpose $\beta(\bar{v})$, $\gamma(\bar{u},\bar{v})$
    achieve $\beta(\bar{v})$ varying $\bar{v}$
    achieve $\gamma(\bar{u},\bar{v})$ varying $\bar{u}$
assert $\beta(\bar{v})$, $\gamma(\bar{u},\bar{v})$ .
```

For example, the two conjuncts of the goal

achieve $x=0$, $z=y$ **varying** x,y,z

contain different variables. We can therefore split it into

```
purpose  $x=0$,  $z=y$
    achieve  $x=0$  varying  $x$
    achieve  $z=y$  varying  $y, z$
assert  $x=0$,  $z=y$ .
```

● **Protection Rule.** Another strategy is to insist that after each stage executed in achieving γ , β remains true for the current values of the variables:

```
achieve  $\beta(\bar{v})$,  $\gamma(\bar{u},\bar{v})$  varying  $\bar{u},\bar{v}$
-----------------------------------------------------
purpose  $\beta(\bar{v})$,  $\gamma(\bar{u},\bar{v})$
    achieve  $\beta(\bar{v})$  varying  $\bar{v}$
    achieve  $\gamma(\bar{u},\bar{v})$  protecting  $\beta(\bar{v})$  varying  $\bar{u},\bar{v}$
assert  $\beta(\bar{v})$,  $\gamma(\bar{u},\bar{v})$ .
```

One way to protect a relation is to insist that its variables do not change value, i.e. $\bar{v}=\bar{v}'$, as in the *disjoint goal rule* above; another method is the formation of a loop, with the protected relation serving as the invariant assertion, as we shall see.

For example,

```
achieve  $z=gcd(s,t)$,  $gcd(s,t)=gcd(a,b)$  varying  $z,s,t$
```

may be broken into

```
achieve  $gcd(s,t)=gcd(a,b)$  varying  $s,t$
achieve  $z=gcd(s,t)$  protecting  $gcd(s,t)=gcd(a,b)$  varying  $z,s,t$ .
```

● **Preservation Rule.** Assume that the program variables $\bar{v}$ are not output variables, rather they were introduced to facilitate achieving some purpose $\alpha(\bar{u})$. Then the final values of $\bar{v}$ are unimportant, and one need only achieve β and γ for some *arbitrary* values of $\bar{v}$. As we saw, the *protection rule* achieves both β and γ for the *final* values of $\bar{v}$; while in the *disjoint rule*, the values of $\bar{v}$ are the same after achieving γ as after achieving β . A third possibility is that after achieving β for some $\bar{v}$, one achieve γ for those *same* values of $\bar{v}$ though the current value of $\bar{v}$ may be changed in the process. The only requirement is that achieving γ for the new values of $\bar{v}$ also implies γ for the previous values of $\bar{v}$. (Equivalently, if γ was the goal for the old $\bar{v}$, then γ remains the goal after this stage — for the new $\bar{v}$.) Thus, by achieving γ , we end up with β and γ holding for the old values of $\bar{v}$.

The rule is

```
purpose $\alpha(\bar{u})$
     achieve $\beta(\bar{v})$,  $\gamma(\bar{u},\bar{v})$ varying $\bar{u},\bar{v}$
________________________________________________________
purpose $\alpha(\bar{u})$
     achieve $\beta(\bar{v})$ varying $\bar{v}$
     achieve $\gamma(\bar{u},\bar{v})$ preserving $\gamma(\bar{u},\bar{v})$ for $\bar{v}$ varying $\bar{u},\bar{v}$
assert $\alpha(\bar{u})$ .
```

The second goal γ may then be transformed further. In particular, this rule can lead to a loop, with the preserved relation serving as the invariant purpose of the loop. We require that γ remain the goal for current values of the variables throughout the achievement of γ itself.

For example,

achieve $z=gcd(s,t)$, $(s,t)=(a,b)$ **varying** z,s,t

may be broken into

achieve $(s,t)=(a,b)$ **varying** s,t
achieve $z=gcd(s,t)$ **preserving** $z=gcd(s,t)$ **for** s,t **varying** z,s,t .

The second subgoal may then be strengthened to

achieve $s=0$, $z=t$ **preserving** $z=gcd(s,t)$ **for** s,t **varying** z,s,t .

To summarize the difference between the last two rules, we may say that the *protection rule* applies to goals *already* achieved, while the *preservation rule* applies to goals *to be* achieved.

5. *Loop Rules*

The loop rules allow a given goal to be achieved step by step. We present two rules for forming iterative loops and a rule for guaranteeing their termination. We also include a recursion-formation rule.

- **Forward Iterative Loop Rule.** Given a goal of the form

```
achieve $\beta$ protecting $\alpha$ varying $\bar{u}$
```

(preceding code has achieved α and we wish to keep α true while achieving β), the following rule will generate an iterative loop:

```
achieve $\beta(\bar{u})$ protecting $\alpha(\bar{u})$ varying $\bar{u}$
----------------------------------------------------------------
purpose $\beta(\bar{u})$, $\alpha(\bar{u})$
     loop   assert $\alpha(\bar{u})$
            until $\beta(\bar{u})$
            approach $\beta(\bar{u})$ protecting $\alpha(\bar{u})$ varying $\bar{u}$
            repeat
assert $\beta(\bar{u})$, $\alpha(\bar{u})$
```

This is permissible provided the exit test β is primitive.

The statement

```
assert $\alpha(\bar{u})$
```

is the invariant assertion of the loop stating that α is true whenever execution reaches the beginning of the loop. It is invariant, since it is given that α is true when the loop is first entered (and only needs to be protected), and the loop-body subgoal

```
approach $\beta(\bar{u})$ protecting $\alpha(\bar{u})$ varying $\bar{u}$
```

will ensure that α remains true after each iteration. The loop will terminate when the exit condition β becomes true; at that point both the invariant α and the test β must hold. In order to guarantee that loop execution will indeed terminate, we must make definite progress towards β ; this is the meaning of "approach".

For example, the goal

```
achieve $s=0$ protecting $gcd(s,t)=gcd(a,b)$ varying $z,s,t$
```

suggests the loop

```
purpose $s=0$, $gcd(s,t)=gcd(a,b)$
     loop assert $gcd(s,t)=gcd(a,b)$
          until $s=0$
          approach $s=0$ protecting $gcd(s,t)=gcd(a,b)$ varying $s,t$
          repeat
assert $s=0$, $gcd(s,t)=gcd(a,b)$ .
```

The new goal

approach $s=0$

can be achieved by decreasing the value of s, provided that s was nonnegative upon loop entry. The invariant must be protected in the process.

● **Backward Iterative Loop Rule.** For the case where we wish to achieve a relation β while preserving an ultimate goal α, we have

```
achieve $\beta(\bar{u},\bar{v})$ preserving $\alpha(\bar{u},\bar{v})$ for $\bar{v}$ varying $\bar{u},\bar{v}$
--------------------------------------------------------------------------------
purpose $\beta(\bar{u},\bar{v})$
     loop    purpose $\alpha(\bar{u},\bar{v})$
             until $\beta(\bar{u},\bar{v})$
             approach $\beta(\bar{u},\bar{v})$ preserving $\alpha(\bar{u},\bar{v})$ for $\bar{v}$ varying $\bar{u},\bar{v}$
             repeat
assert $\beta(\bar{u},\bar{v})$ .
```

As with the forward loop, this is permissible only if β is computable.

The purpose of the loop is to achieve the exit relation β while preserving the ultimate purpose α. The loop will terminate when the exit condition β becomes true; at that point, the fact that β holds may be used to help achieve the purpose α. The statement

purpose $\alpha(\bar{u},\bar{v})$

contains the invariant purpose of the loop and states that whenever execution reaches the beginning of the loop, what remains to be computed is α, for the current values of the variables $\bar{u}$ and $\bar{v}$. Upon exiting the loop, the goal is α, and α is the goal whenever the loop-body subgoal

approach $\beta(\bar{u},\bar{v})$ **preserving** $\alpha(\bar{u},\bar{v})$ **for** $\bar{v}$ **varying** $\bar{u},\bar{v}$

is executed.

For example, the goal

achieve $s=0$, $z=t$ **preserving** $z=gcd(s,t)$ **for** s,t **varying** z,s,t ,

may be split into disjoint goals

```
achieve $s=0$ preserving $z=gcd(s,t)$ for $s,t$ varying $s,t$
assert $s=0$
purpose $z=t$
achieve $z=t$ preserving $z=gcd(s,t)$ for $s,t$ varying $z$ .
```

By assigning $z:=t$, the subgoal $z=t$ is achieved as is the preserved goal $z=gcd(s,t)$, since $s=0$. Letting α be $z=gcd(s,t)$ and β be $s=0$, the remaining subgoal

achieve $s=0$ **preserving** $z=gcd(s,t)$ **for** s,t **varying** s,t

may be transformed into the loop

```
purpose $s=0$
    loop purpose $z=gcd(s,t)$
        until $s=0$
        approach $s=0$ preserving $z=gcd(s,t)$ for $s,t$ varying $s,t$
        repeat
assert $s=0$ .
```

Within the loop, the purpose $z=gcd(s,t)$ must be maintained while making progress towards $s=0$.

● **Termination Rule.** Assume that we are given a loop-body subgoal

approach $\beta(\bar{u})$ **protecting** $\alpha(\bar{u})$ **varying** $\bar{u}$.

Clearly in order to make progress towards β , one of the variables $\bar{u}$ must be changed, i.e. $\bar{u}\neq\bar{u}'$. This is not however sufficient to ensure that β will ever be attained. What we need is the notion of well-founded set: a *well-founded set* $(W,\succ)$ consists of a set of elements W and an ordering $\succ$ defined on the elements, such that there can be no infinite descending sequences of elements $w_1\succ w_2\succ\ldots$. So, if throughout execution of the loop we keep $\bar{u}\in W$, for some well-founded set $(W,\succ)$, and insist that with each iteration $\bar{u}$ is reduced in that ordering, i.e. $\bar{u}'\succ\bar{u}$, then termination is guaranteed. In particular, we must have $\bar{u}_0\succ\bar{u}_*$, where $\bar{u}_0$ denotes the value of $\bar{u}$ upon entering the loop and $\bar{u}_*$ denotes the value upon exiting. (To determine this, we may use whatever facts are known about $\bar{u}_0$ and $\bar{u}_*$, e.g. $\alpha(\bar{u}_0)$, $\alpha(\bar{u}_*)$, and $\beta(\bar{u}_*)$.)

We have, for forward loops, the termination rule

$$\frac{\begin{array}{l}\textbf{assert}\ \bar{u}_0,\bar{u}_*\in W,\ \bar{u}_0\succ\bar{u}_* \\ \textbf{approach}\ \beta(\bar{u})\ \textbf{protecting}\ \alpha(\bar{u})\ \textbf{varying}\ \bar{u}\end{array}}{\begin{array}{l}\textbf{assert}\ \neg\beta(\bar{u}) \\ \textbf{achieve}\ \bar{u}'\succ\bar{u}\ \textbf{protecting}\ \alpha(\bar{u}),\ \bar{u}\in W\ \textbf{varying}\ \bar{u}\end{array}}$$

and similarly for backward loops

```
assert $\bar{u}_0, \bar{u}_* \in W,\ \bar{u}_0 \succ \bar{u}_*$
approach $\beta(\bar{u})$ preserving $\alpha(\bar{u})$ for $\bar{v}$ varying $\bar{u}$
--------------------------------------------------
assert $\neg\beta(\bar{u})$
achieve $\bar{u}' \succ \bar{u}$ preserving $\alpha(\bar{u}),\ \bar{u} \in W$ for $\bar{v}$ varying $\bar{u}$ ,
```

where $(W, \succ)$ is some well-founded set.

The well-founded set most commonly used for termination proofs is the set $\mathbb{N}$ of nonnegative integers under the $>$ ordering. When dealing with more than one variable, the lexicographic ordering on n-tuples is useful. For example, to use the lexicographic ordering for two variables u and v, we first look for well-founded sets W_1 and W_2 such that $u \in W_1$ and $v \in W_2$. Then we consider the pair (u, v) and require

achieve $(u', v') \succ (u, v)$ **protecting** $\alpha(u, v),\ u \in W_1,\ v \in W_2$ **varying** u, v .

where $\succ$ is the lexicographic ordering on pairs, i.e. we must

achieve $u' \succ u \vee (u' = u \wedge v' \succ v)$ **protecting** $\alpha(u, v),\ u \in W_1,\ v \in W_2$
varying u, v .

Another, often useful, way of handling several variables is based on an assumption of monotonicity for each variable. Determining, for some variable u, that the initial value u_0 is greater than the final value u_* suggests that u decrease monotonically, i.e. $u' \succ u$. In that case, we may also

assert $u_0 \succ u \succ u_*$

within the loop (provided we can determine the values of u_0 and u_*). Given two monotonic variables u and v, termination may be ensured by requiring

achieve $u' \succ u,\ v' \succ v,\ u' \succ u \vee v' \succ v$
protecting $\alpha(u, v),\ u \in W_1,\ v \in W_2,\ u \succ u_*,\ v \succ v_*$ **varying** u_1, v

In other words, each iteration reduces the value of one of the variables, without increasing any other (cf. the multiset ordering on $\{u, v\}$ in Dershowitz and Manna [Mar. 1978]).

- **Recursive Loop Rule.** The following rule forms a recursive loop:

```
assert  $\varphi(\bar{u}), \bar{u}\in W$
purpose  $\alpha(\bar{u})$
      . . .
      assert  $\varphi(\bar{v}),\ \bar{v}\in W,\ \bar{u}\succ\bar{v}$
      achieve  $\alpha(\bar{v})$  varying  $\bar{v}$
      . . .
assert  $\alpha(\bar{u})$
________________________________________
assert  $\varphi(\bar{u}), \bar{u}\in W$
$P(\bar{u})$: begin  purpose  $\alpha(\bar{u})$
      . . .
      assert  $\varphi(\bar{v}),\ \bar{v}\in W,\ \bar{u}\succ\bar{v}$
      purpose  $\alpha(\bar{v})$
               $P(\bar{v})$
      assert  $\alpha(\bar{v})$
      . . .
      end
assert  $\alpha(\bar{u})$ .
```

The current goal is

achieve $\alpha(\bar{v})$ **varying** $\bar{v}$,

while the code that is being synthesized — call it P — has the similar

purpose $\alpha(\bar{u})$.

Before we can insert a recursive call $P(\bar{v})$, we must know that the input assertion φ is satisfied by the arguments $\bar{v}$. Furthermore, to guarantee that the recursion will not continue forever, we require $\bar{u}\succ\bar{v}$ in some well-founded ordering $(W, \succ)$.

Conditional-formation techniques are the subject of Luckham and Buchanan [1974] and Warren [1976]. The achievement of conjunctive goals is the topic of Waldinger [1977]; protection mechanisms are used for this purpose by Sussman [1975]; Sacerdoti [1975] addresses their nonlinear nature. The use of invariants for the automatic construction of iterative loops is also discussed by Duran [1975]. Recursion-formation techniques are discussed in detail by Manna and Waldinger [1977]; similar work appears in Siklossy [1974] and Darlington [1975]. Misra [1975] gives criteria for a loop to be formed directly from the specifications.

In the next section, we apply these rules to the synthesis of several programs.

4. EXAMPLES

Our first example is a straightforward synthesis of the integer square-root function. Arrays are introduced in the second example, which is a program to find the position of a minimal element of an array. Our concluding example is Hoare's Partition algorithm [1961]; it is a nontrivial problem, requiring some degree of understanding and ingenuity to program.

Example 1: *Integer Square-root.*

In an earlier chapter we developed a binary integer square-root program from a schema; in this chapter our goal is to synthesize some program satisfying the specifications

```
P_1: begin comment integer square-root program
     assert $a \in \mathbb{N}$
     achieve $z = \lfloor \sqrt{a} \rfloor$ varying $z$
     end
```

from scratch. The program should set the variable z to the largest integer not greater than the square-root of a, for any nonnegative integer a.

We assume that the $\sqrt{\ }$ function is not primitive; otherwise we could achieve our goal using the *assignment rule* to obtain

$$z := \lfloor \sqrt{a} \rfloor .$$

Therefore, as a first step, we endeavor to replace the goal with one that does not contain the $\sqrt{\ }$ function.

Using the definition of $\lfloor u \rfloor$,

fact $v = \lfloor u \rfloor \equiv v \le u \wedge u < v+1 \wedge v \in \mathbb{Z}$,

the goal

achieve $z = \lfloor \sqrt{a} \rfloor$ **varying** z

may be transformed into the equivalent goal

```
purpose $z=\lfloor\sqrt{a}\rfloor$
    achieve $z\le\sqrt{a}$, $\sqrt{a}<z+1$, $z\in Z$ varying $z$
assert $z=\lfloor\sqrt{a}\rfloor$ .
```

Using the

fact $u\le\sqrt{v} \equiv u^2\le v$ **when** $u\ge0$

to eliminate the $\sqrt{\ }$ operator, the conjunct $z\le\sqrt{a}$ may be replaced by $z^2\le a$ and $\sqrt{a}<z+1$ may be replaced by $a<(z+1)^2$, with the side conditions $z\ge0$ and $z+1\ge0$ added:

achieve $z^2\le a$, $z\ge0$, $a<(z+1)^2$, $z+1\ge0$, $z\in Z$ **varying** z .

This simplifies to just

achieve $z^2\le a$, $a<(z+1)^2$, $z\in N$ **varying** z .

The above subgoal is a conjunction of three relations; the *protection rule* suggests splitting it into two consecutive subgoals:

```
purpose $z^2\le a$, $a<(z+1)^2$, $z\in N$ varying $z$
    achieve $a<(z+1)^2$, $z\in N$ varying $z$
    achieve $z^2\le a$ protecting $a<(z+1)^2$, $z\in N$ varying $z$
assert $z^2\le a$, $a<(z+1)^2$, $z\in N$ varying $z$ .
```

Later, we shall see what alternative splittings might result in.

To achieve the first subgoal

achieve $a<(z+1)^2$, $z\in N$ **varying** z

we apply the *strengthening rule* to this subgoal, using the transitivity of inequality expressed in the

fact $u<w$ **when** $u<v$, $v\le w$,

obtaining the stronger

achieve $a<v$, $v\le(z+1)^2$, $z\in N$ **varying** z,v ;

Now, the

fact $u\le u^2$ **when** $u\ge1\vee u\le0$

tells us that taking $z+1$ for v will give $v \le (z+1)^2$, provided that $z+1 \ge 1 \vee z+1 \le 0$. The subgoal $z \in N$ implies $z+1 \ge 1$, leaving

```
        achieve $a<z+1$, $z \in N$ varying $z$ .
```

Since v is not an output variable, it has been eliminated from the goal. The goal may be strengthened further to

```
        achieve $a=z$ varying $z$,
```

by matching it with the

```
        fact $u<u+v$ when $v>0$ .
```

This goal, in turn, may be attained by the simple assignment

```
        purpose $a<(z+1)^2$, $z \in N$
            $z := a$
        assert $z=a$ .
```

The *forward loop rule suggests* turning the second subgoal

```
        achieve $z^2 \le a$ protecting $a<(z+1)^2$, $z \in N$ varying $z$
```

into a loop with the invariant

```
        assert $a<(z+1)^2$, $z \in N$
```

maintained true until the exit clause

```
        until $z^2 \le a$
```

becomes true. We have the skeleton of a loop:

```
        purpose $a<(z+1)^2$, $z \in N$
            $z := a$
        assert $z=a$
        purpose $z^2 \le a$, $a<(z+1)^2$, $z \in N$
            loop assert $a<(z+1)^2$, $z \in N$
                until $z^2 \le a$
                approach $z^2 \le a$ protecting $a<(z+1)^2$, $z \in N$ varying $z$
                repeat
        assert $z^2 \le a$, $a<(z+1)^2$, $z \in N$ .
```

Within the loop body, we must

approach $z^2 \le a$ **protecting** $a < (z+1)^2$, $z \in \mathbb{N}$ **varying** z

in order to make progress towards the exit test, while protecting the invariants.

To ensure termination of this loop, the *termination rule* requires that the nonnegative integer z be reduced in some well-founded ordering. We note that upon exit $z_*^2 \le a$, while upon entering the loop $z_0 = a$. Therefore, $z_* \le z_*^2 \le z_0$, and we hypothesize that z is decreasing monotonically from a to it final value. We therefore take the set of nonnegative integers $\mathbb{N}$ under the usual $>$ ordering as the well-founded set. We have obtained the loop-body subgoal

assert $a < z^2$
achieve $z' > z$ **protecting** $a < (z+1)^2$, $z \in \mathbb{N}$ **varying** z ,

i.e. we wish to set the nonnegative integer z to a value less than its current one, while protecting the loop invariant $a < (z+1)^2$. The assertion indicates that the exit test $z^2 \le a$ does not yet hold if the loop is being continued.

With each loop iteration we wish to decrease the value of z, while protecting the invariant $a < (z+1)^2$. Using the transitivity of inequality again, suggests looking for some v such that $a < v$ and $v \le (z+1)^2$. But $a < z'^2$ may be asserted for the previous value of z; therefore, to achieve $a < (z+1)^2$, we need only achieve $z'^2 \le (z+1)^2$, i.e. $z' \le z+1$. This leaves us with the goal

achieve $z' > z$, $z' \le z+1$ **protecting** $z \in \mathbb{N}$ **varying** z

which is equivalent to

achieve $z = z' - 1$ **varying** z

and may be achieved by the assignment

$z := z - 1$.

We have derived the program

```
P_1: begin comment an integer square-root program
     assert $a \in N$
     $z := a$
     loop assert $a < (z+1)^2,\ z \in N$
          until $z^2 \le a$
          $z := z-1$
          repeat
     assert $z = \lfloor \sqrt{a} \rfloor$
     end .
```

With most of the subgoals left in, the program would look like:

```
P_1: begin comment a cluttered integer square-root program
     assert $a \in N$
     purpose $z = \lfloor \sqrt{a} \rfloor$
        purpose $z \le \sqrt{a},\ \sqrt{a} < z+1,\ z \in Z$
           purpose $z^2 \le a,\ a < (z+1)^2,\ z \in N$
              purpose $a < (z+1)^2,\ z \in N$
                 $z := a$
              assert $z = a$
              purpose $z^2 \le a,\ a < (z+1)^2,\ z \in N$
                 loop assert $a < (z+1)^2,\ z \in N$
                        until $z^2 \le a$
                              purpose $z' > z,\ a < (z+1)^2,\ z \in N$
                                    purpose $z' > z,\ z' \le z+1,\ z \in N$
                                          $z := z-1$
                                    assert $z' > z,\ z' \le z+1,\ z \in N$
                              assert $z' > z,\ a < (z+1)^2,\ z \in N$
                        repeat
              assert $z^2 \le a,\ a < (z+1)^2,\ z \in N$
           assert $z^2 \le a,\ a < (z+1)^2,\ z \in N$
        assert $z \le \sqrt{a},\ \sqrt{a} < z+1,\ z \in Z$
     assert $z = \lfloor \sqrt{a} \rfloor$
     end .
```

What would have happened had we split the goal

achieve $z^2 \le a,\ a<(z+1)^2,\ z \in N$ **varying** z

in a different manner? Had we chosen to first

achieve $z^2 \le a,\ z \in N$ **varying** z

and then

achieve $a<(z+1)^2$ **protecting** $z^2 \le a,\ z \in N$ **varying** z ,

we would be led in a similar manner to the loop skeleton

```
achieve $z^2 \le a,\ z \in N$ varying $z$
purpose $z^2 \le a,\ a<(z+1)^2,\ z \in N$
    loop assert $z^2 \le a,\ z \in N$
        until $a<(z+1)^2$
        approach $a<(z+1)^2$ protecting $z^2 \le a,\ z \in N$ varying $z$
        repeat
assert $z^2 \le a,\ a<(z+1)^2,\ z \in N$ .
```

To achieve the initialization subgoal, we note that the input assertion $a \in N$ implies $a \ge 0$. Therefore, to achieve $z^2 \le a$, it suffices for $z^2 \le 0$. But the

fact $0 \le u^2$

implies that $z^2=0$, so we may initialize $z:=0$. We would then decide to approach the exit test by *increasing* z from 0 . To guarantee termination, we could use the well-founded set of integers less than $\sqrt{a}$; the smaller the integer, the greater it is in the well-founded ordering. Continuing in a manner paralleling the derivation of P_1 , we get

```
P_1': begin comment alternative integer square-root program
    assert $a \in N$
    $z := 0$
    loop $z^2 \le a,\ z \in N$
        until $a < (z+1)^2$
        $z := z+1$
        repeat
    assert $z = \lfloor \sqrt{a} \rfloor$
    end .
```

In the section on extension, we shall see how this program may be improved.

Had we split the goal

```
    achieve $z \le \sqrt{a},\ \sqrt{a} < z+1,\ z \in N$ varying $z$
```

into

```
    achieve $z \in N$ varying $z$
    achieve $z^2 \le a,\ a < (z+1)^2$ protecting $z \in N$ varying $z$ ,
```

we would be led to

```
    assert $a \in N$
    achieve $z \in N$ varying $z$
    loop assert $z \in N$
        until $z^2 \le a \wedge a < (z+1)^2$
        approach $z^2 \le a \wedge a < (z+1)^2$ protecting $z \in N$ varying $z$
        repeat
    assert $z = \lfloor \sqrt{a} \rfloor$ .
```

The choice of initial value for z such that $z \in N$ is completely arbitrary; to ensure termination we would have to first determine whether the chosen initial value is less or greater than the desired final value.

The resulting program would be

```
assert $a \in N$
$z :\in N$
loop assert $z \in N$
    until $z^2 \le a \wedge a < (z+1)^2$
    if $z^2 \le a$ then $z := z+1$ else $z := z-1$ fi
    repeat
assert $z = \lfloor \sqrt{a} \rfloor$ ,
```

where $z :\in N$ is a nondeterministic assignment of some element of the set N to the variable z. This solution is more complicated than either of the previous two possibilities. In general, it is advisable to maintain invariant as much of the goal as possible and keep the exit test as simple as possible.

Example 2: *Array Minimum.*

In this example, we wish to synthesize a program to search for the position of a minimal element in an array segment. Our goal is to synthesize a program for

```
$\cdot P_2$: begin comment array minimum-position program
    assert $i, j \in N,\ i \le j$
    achieve $A[z] \le A[i:j],\ i \le z \le j$ varying $z$
    end .
```

The conjunct $A[z] \le A[i:j]$ is short for $(\forall \zeta)(i \le \zeta \le j) A[z] \le A[\zeta]$; in general, for any predicate p , $p[u:v]$ is short for $(\forall \zeta)(u \le \zeta \le v) p(\zeta)$. Note that the array A is constant and only the value of the variable z may be altered by the program. In other words, we wish to set the variable z to the index of an occurrence of the smallest element in the nonempty array segment $A[i:j]$.

Using the

fact $p(\bar{u})$ **when** $p(\bar{v}),\ \bar{u} = \bar{v}$,

this goal may be strengthened by introducing a new program variable y and substituting it for the constant i ; the conjunct $y = i$ must be added to the goal. Introducing a new variable will allow the program to manipulate its value so that the goal may be achieved in stages. We derive

```
purpose $A[z] \le A[i:j]$, $i \le z \le j$
    achieve $A[z] \le A[y:j]$, $y \le z \le j$, $y=i$ varying $z,y$
assert $A[z] \le A[i:j]$, $i \le z \le j$ .
```

Using the *protection rule*, we shall attempt to achieve this goal in two stages, first achieving both $A[z] \le A[y:j]$ and $y \le z \le j$ and then achieving $y=i$:

```
achieve $A[z] \le A[y:j]$, $y \le z \le j$ varying $z,y$
achieve $y=i$ protecting $A[z] \le A[y:j]$, $y \le z \le j$ varying $z,y$ .
```

By matching the first goal

achieve $A[z] \le A[y:j]$, $y \le z \le j$ **varying** z,y

with the

fact $p[u:u]$ **when** $p(u)$,

the goal may be strengthened to

achieve $A[z] \le A[j]$, $y=j$, $y \le z \le j$ **varying** z,y .

Using reflexivity

fact $u \le u$

to further strengthen this goal, we get

achieve $A[z]=A[j]$, $y=j$, $y \le z \le j$ **varying** z,y .

Now the

fact $f(u)=f(v)$ **when** $u=v$,

for any function f (the array A may be considered a function), suggests

achieve $z=j$, $y=j$ **varying** z,y .

This is in turn achievable by the multiple assignment

$(z,y) := (j,j)$

We are left with the subgoal

achieve $y=i$ **protecting** $A[z] \le A[y:j]$, $y \le z \le j$ **varying** z,y ,

which, by the *forward-loop rule*, suggests the iterative loop

```
loop assert $A[z]\le A[y:j]$, $y\le z\le j$
     until $y=i$
     approach $y=i$ protecting $A[z]\le A[y:j]$, $y\le z\le j$ varying $z,y$
     repeat
assert $A[z]\le A[y:j]$, $y\le z\le j$, $y=i$ .
```

The remaining loop-body subgoal is

```
approach $y=i$ protecting $A[z]\le A[y:j]$, $y\le z\le j$ varying $z,y$ .
```

By noting that upon entering the loop $y_0=j$ and upon exiting the loop $y_*=i$, where it is given that $i\le j$, the *termination rule* suggests that the variable y remain an integer and decrease monotonically from j to i. Including the range of y, we now have

```
assert $y\ne i$
achieve $y'>y$ protecting $A[z]\le A[y:j]$, $y\le z\le j$, $y\in Z$, $i\le y\le j$ varying $z,y$
```

In other words, assuming that the goal $y=i$ has not yet been achieved, we wish to decrease y while protecting the invariants $A[z]\le A[y:j]$ and $y\le z\le j$ along with the added invariants $y\in Z$ and $i\le y\le j$ for termination.

Since we are assuming that $y'\in Z$ and $i\le y'$ hold, and we know that for the loop to be continued $y'\ne i$, it follows that $i\le y'-1$. So, in order to achieve $i\le y$, we need to achieve $y'-1\le y$. This, together with the additional requirement that $y<y'$ and $y\in Z$, forces $y=y'-1$. After assigning

$$y := y-1 ,$$

the remaining goal is

```
assert $A[z]\le A[y+1:j]$, $y+1\le z\le j$
achieve $A[z]\le A[y:j]$, $y\le z\le j$ varying $z$ .
```

Since the value of y has changed, we have broken the protected clause into an assertion that the conjuncts held for the previous value of y and z and a goal to reachieve them for $y-1$. The assignment to y is protected by only varying z in this goal.

Part of the above goal has already been achieved and part remains to be achieved. Using the following basic fact about universal quantifiaction:

```
fact $p[u:v]$ when $p[u:w]$, $p[w+1:v]$, $u\le w\le v$ ,
```

we can break the conjunct $A[z]\le A[y:j]$ into two parts:

```
purpose $A[z]\le A[y:j]$, $y\le z\le j$
     achieve $A[z]\le A[y:w]$, $A[z]\le A[w+1:j]$, $y\le w\le j$, $y\le z\le j$ varying $z,w$
assert $A[z]\le A[y:j]$, $y\le z\le j$
```

Since we have already asserted $A[z'] \le A[y+1:j]$ and $y < y+1 \le z' \le j$, we may achieve $A[z] \le A[w+1:j]$, $y \le w \le j$, and $y \le z \le j$ by leaving z unchanged and letting $w=y$. We are left with only

```
achieve $A[z] \le A[y]$ .
```

This latest goal has an empty variable list; it can only be achieved by proving that it is true or testing that it is true. Since it cannot be proved true, we use the *conditional rule* to generate

```
if $A[z] \le A[y]$   then
                     else  assert $A[z] \le A[y+1:j]$, $y+1 \le z \le j$, $A[y] < A[z]$
                           achieve $A[z] \le A[y:w]$, $A[z] \le A[w+1:j]$, $y \le w \le j$,
                                                        $y \le z \le j$ varying $z, w$ .
                     fi .
```

The **then**-clause is empty, since $A[z] \le A[y]$ holds at that point by virtue of the test; when the conditional test is false, we may use that fact, along with the fact that the invariants held for the prior value of y to achieve the previous goal. We know, then, that $A[y] < A[z'] \le A[y+1:j]$, so to achieve $A[z] \le A[w+1:j]$ we let $z=y=w$. Substituting for z and w, we have

```
achieve $z=y=w$, $A[y] \le A[y:y]$, $A[y] \le A[y+1:j]$, $y \le y \le j$ varying $z, w$ .
```

Since $A[z] \le A[y+1:j]$ and $y+1 \le j$ have already been asserted, this reduces to just

```
achieve $z=y$ varying $z$ .
```

Achieving this via an assignment statement, we obtain the conditional

```
if $A[z] \le A[y]$ then else $z := y$ fi ,
```

or simply

```
if $A[y] < A[z]$ then $z := y$ fi .
```

We have derived this program for minimum:

```
P₂: begin comment array minimum-position program
    assert $i,j \in \mathbb{N}$, $i \le j$
    $(z,y)$ := $(j,j)$
    loop assert $A[z] \le A[y{:}j]$, $y \in \mathbb{Z}$, $i \le y \le z \le j$
        until $y = i$
        $y$ := $y-1$
        if $A[y] < A[z]$ then $z$ := $y$ fi
        repeat
    assert $A[z] \le A[i{:}j]$, $i \le z \le j$
    end .
```

In the next section, we shall see how to extend this program to achieve the added relation $x = A[z]$.

Example 3: *Partition.*

In this last synthesis example, we consider the Partition problem: given an array segment $A[i{:}j]$, rearrange its elements so that there exists some position g which partitions the segment into two ordered parts. In other words, each element of the left part $A[i{:}g]$ is to be less than or equal to each element of the right part $A[g+1{:}j]$. The goal specification may be expressed as

```
P₃: begin comment partition program
    assert $i,j \in \mathbb{N}$, $i < j$
    achieve $A[i{:}g] \le A[g+1{:}j]$, $i \le g$, $g+1 \le j$, $bag(A[i{:}j]) = bag(A'[i{:}j])$ varying $A, g$
    end ,
```

where A' represents the prior value of the array A . The function $bag(A[u{:}v])$ yields the multiset $\{A[u], A[u+1], \ldots, A[v]\}$; thus, the fourth conjunct of the goal implies that the new array segment must be a permutation of the original segment. We illustrate two possible solutions.

1. *First Solution*

As a first try, we strengthen the goal specification using the

```
    fact  $p[u{:}u]$  when  $p(u)$
```

to eliminate the quantifier $[i{:}g]$. What we wish, then, is to

```
    achieve  $A[i]\le A[g+1{:}j]$, $g=i$, $bag(A[i{:}j])=bag(A'[i{:}j])$  varying  $A, g$
```

(the subgoals $i\le g$ and $g+1\le j$ were deleted since they follow from the new goal $g=i$ and the assertion $i<j$.) The subgoal $g=i$ can be achieved by the assignment

```
    $g := i$ ,
```

leaving only

```
    achieve  $A[i]\le A[i+1{:}j]$, $bag(A[i{:}j])=bag(A'[i{:}j])$  varying  $A$ .
```

We have already seen how to synthesize a program to find the position of a minimal element of an array; so we know how to

```
    achieve  $A[z]\le A[i+1{:}j]$, $i+1\le z\le j$  varying  $z$ .
```

This, along with the transitivity of inequality,

```
    fact  $u\le v$  when  $u\le w$, $w\le v$ ,
```

suggest strengthening the above goal to

```
    achieve  $A[i]\le w$, $w\le A[i+1{:}j]$, $bag(A[i{:}j])=bag(A'[i{:}j])$  varying  $A, w$ ,
```

where the new program variable w can be set to any convenient value. Comparing what we have with what we want suggests letting $w=A[z]$ and splitting the goal into the disjoint goals

```
    achieve  $A[z]\le A[i+1{:}j]$, $i+1\le z\le j$, $bag(A[i{:}j])=bag(A'[i{:}j])$  varying  $z$
    achieve  $A[i]\le A[z]$  protecting  $A[z]\le A[i+1{:}j]$, $i+1\le z\le j$,
                                     $bag(A[i{:}j])=bag(A'[i{:}j])$  varying  $A$ .
```

The permutation requirement in the first goal is satisfied, since that goal does not vary A , i.e. $A'=A$; the rest is achieved by the old minimum program. To achieve the second goal $A[i]\le A[z]$, one might try to assign $A[z]:=A[i]$, but that would not protect the permutation requirement. We can however test if $A[i]\le A[z]$ already holds:

```
    if  $A[i]\le A[z]$     then
                         else  assert  $A[z]<A[i]$
                               achieve  $A[i]\le A[z]$  protecting  ...  varying  $A$
                         fi .
```

Knowing, for the else-branch, that $A'[z] \le A'[i]$ suggests achieving $A[i] \le A[z]$ by letting $A'[z] = A[i]$ and $A'[i] = A[z]$. Since $i \le z \le j$, this also protects the permutation requirement $bag(A[i:j]) = bag(A'[i:j])$. We have

```
if  $A[i] \le A[z]$     then
                        else  $(A[i], A[z]) := (A[z], A[i])$
                        fi .
```

Accordingly, our first solution is

```
P₃: begin  comment  first partition program
    assert  $i, j \in N$,  $i < j$
    $g := i$
    $(z, y) := (j, j)$
    loop assert  $A[z] \le A[y:j]$,  $y \le z \le j$
         until  $y = i+1$
         $y := y-1$
         if  $A[y] < A[z]$  then  $z := y$  fi
         repeat
    if  $A[z] < A[i]$  then  $(A[i], A[z]) := (A[z], A[i])$  fi
    assert  $A[i:g] \le A[g+1:j]$,  $i \le g$,  $g+1 \le j$,  $bag(A[i:j]) = bag(A'[i:j])$
    end .
```

This program leaves something to be desired. Despite the fact that it satisfies the *stated* specifications and that it is not inefficient, the fact that for the usual applications of Partition it is desirable that g be closer to the mean of i and j went unspecified. The above program always results in g being equal to i.

2. *Second Solution*

We do not want, then, to force $g=i$. Instead, we leave g variable and reconsider our original goal (temporarily leaving out the permutation requirement)

achieve $A[i:g] \le A[g+1:j]$, $i \le g$, $g+1 \le j$ **varying** A, g .

This time, we first strengthen the goal by introducing a new variable h to replace the expression $g+1$:

achieve $A[i:g] \le A[h:j]$, $g+1=h$, $i \le g$, $h \le j$ **varying** A, g, h .

Then we try to eliminate the double quantifier in $A[i:g] \le A[h:j]$ by introducing a new variable w, using the

fact $u \le v$ **when** $u \le w,\ w \le v$.

This yields

achieve $A[i:g] \le w,\ w \le A[h:j],\ g+1=h,\ i \le g,\ h \le j$ **varying** A, g, h, w .

There are now four variables that the program may set: A , g , h , and w .

Now we can split this conjunctive goal into two:

achieve $A[i:g] \le w,\ w \le A[h:j],\ i \le g,\ h \le j$ **varying** A, g, h, w
achieve $g+1=h$ **protecting** $A[i:g] \le w,\ w \le A[h:j],\ i \le g,\ h \le j$
varying A, g, h, w ;

the second will become a loop with exit test $g+1=h$ and the first will initialize the invariants. By reducing quantifiers to single elements, the first goal may be strengthened to

achieve $A[i] \le w,\ w \le A[j],\ g=i,\ h=j$ **varying** A, g, h, w ,

and the first conjunct may be further strengthened to $A[i]=w$:

achieve $A[i]=w,\ w \le A[j],\ g=i,\ h=j$ **varying** A, g, h, w .

At this point, we would like to assign to w , g , and h . Before we can do that we must substitute for the other occurrence of w :

achieve $A[i]=w,\ A[i] \le A[j],\ g=i,\ h=j$ **varying** A, g, h, w .

Splitting this into two disjoint goals and assigning we get (putting the permutation requirement back in):

achieve $A[i] \le A[j],\ bag(A[i:j])=bag(A'[i:j])$ **varying** A
$(g, h, w) := (i, j, A[i])$.

As in the first solution, the remaining subgoal yields the conditional

if $A[i] > A[j]$ **then** $(A[i], A[j]) := (A[j], A[i])$ **fi** .

The current status of the program is:

```
assert $i,j \in N$,  $i<j$
if $A[i]>A[j]$ then $(A[i],A[j]) := (A[j],A[i])$ fi
$(g,h,w) := (i,j,A[i])$
assert $w=A[i]$,  $A[i]\le A[j]$,  $g=i$,  $h=j$
loop assert $A[i:g]\le w$,  $w\le A[h:j]$,  $i\le g$,  $h\le j$
     until $g+1=h$
     approach $g+1=h$ protecting $A[i:g]\le w$,  $w\le A[h:j]$,  $i\le g$,  $h\le j$
                                                          varying $A,g,h,w$
     repeat .
```

We may now determine bounds for g and h and apply the *termination rule*. Initially $g_0=i<j=h_0$, while upon termination $i\le g_*+1=h_*\le j$. This suggests keeping $g,h\in Z$ and letting g increase from i to its final value g_*, while h decreases from j to h_*. The resulting bounds, $i\le g\le g_*$ and $h_*\le h\le j$, combined with $g_*=h_*-1<h_*$ implies the invariant $g<h$. So to ensure termination, we require that g and h remain integers, and that progress is made by increasing g and/or decreasing h, until they meet somewhere in the middle. Accordingly, the loop-body subgoal becomes

```
achieve $g'\le g$,  $h'\ge h$,  $g'<g \lor h'>h$
     protecting $A[i:g]\le w$,  $w\le A[h:j]$,  $g,h\in Z$,  $i\le g<h\le j$ varying $A,g,h,w$
```

Splitting the two quantifiers into the range that has already been achieved and the range that remains to be achieved, we get

```
achieve $g'\le g$,  $h'\ge h$,  $g'<g \lor h'>h$,  $A[g'+1:g]\le w$,  $w\le A[h:h'-1]$
     protecting $A[i:g]\le w$,  $w\le A[h:j]$,  $g,h\in Z$,  $i\le g<h\le j$ varying $A,g,h,w$ .
```

We shall protect $A[i:g]\le w$ and $w\le A[h:j]$ by not varying w or any of the elements in the array segments $A[i:g]$ and $A[h:j]$. Now if we reduce the quantifier $[g'+1:g]$ to a single element (letting $g'+1=g$) and make the quantifier $[h:h'-1]$ vacuosly true (insisting that $h>h'-1$), then we get

```
achieve $g'\le g$,  $h'\ge h$,  $g'<g \lor h'>h$,  $A[g'+1]\le w$,  $g'+1=g$,  $h>h'-1$,
     protecting $A[i:g]\le w$,  $w\le A[h:j]$,  $g,h\in Z$,  $i\le g<h\le j$ varying $A,g,h$ ,
```

which simplifies to

```
achieve $g'+1=g$,  $h'=h$,  $A[g'+1]\le w$
                        protecting $A[i:g]\le w$,  $w\le A[h:j]$ varying $A,g,h$ .
```

The *conditional rule* suggests achieving the conjunct $A[g'+1]\le w$ by testing:

```
if $A[g+1] \le w$    then  $g := g+1$
                     else  assert $w < A[g]$
                           achieve $g' \le g$, $h' \ge h$, $g' < g \vee h' > h$, $A[g'+1:g] \le w$,
                              $w \le A[h:h'-1]$ protecting . . . varying $g, h$ .
               fi .
```

Without going into more detail, the remaining subgoal generates two more cases: if $w \le A[h'-1]$, then h is decremented by 1 ; otherwise, $A[h'-1] < w < A[g'+1]$ and $A[h'-1]$ is exchanged with $A[g'+1]$ and both g is increased and h decreased. The completed program is:

```
$P_1'$: begin comment partition program
    assert $i, j \in N$, $i < j$
    if $A[i] > A[j]$ then $(A[i], A[j]) := (A[j], A[i])$ fi
    $(g, h, w) := (i, j, A[i])$
    loop assert $A[i:g] \le w \le A[h:j]$, $g, h \in Z$, $i \le g < h \le j$
        until $g+1=h$
        if $A[g+1] \le w$ then $g := g+1$
                          else if $w \le A[h-1]$
                               then $h := h-1$
                               else $(g, h) := (g+1, h-1)$
                                    $(A[g], A[h]) := (A[h], A[g])$
                               fi
                          fi
        repeat
    assert $A[i:g] \le A[g+1:j]$, $i \le g$, $g+1 < j$
    end .
```

5. EXTENSION

In this section, we illustrate techniques for extending a given program to achieve an additional relation. There are two basic methods. One is to append code at the end of the program with the purpose of achieving the additional goal, while making sure that the relations already achieved by the program remain intact. The second method is to achieve the added relation at the outset and modify the program to ensure that it maintains that relation true until the end of the execution.

This second method is also used for local optimization: if a program contains an expression that is relatively difficult to compute, but must be recomputed for each loop iteration, then it may be possible to introduce a program variable that will invariantly contain the value of the complex expression, and for which there is a relatively simple way of deriving the new value of the variable from the old value. This new variable must be updated whenever the value of a variable in the expression is changed, and may be substituted for that expression wherever it occurs in the program text.

Example 1: *Array Minimum.*

Consider our program

```
P_2: begin comment array minimum-position program
    assert $i,j \in \mathbb{N}$, $i \le j$
    $(z,y) := (j,j)$
    loop assert $A[z] \le A[y{:}j]$, $y \le z \le j$
        until $y=i$
        $y := y-1$
        if $A[y] < A[z]$ then $z := y$ fi
        repeat
    assert $A[z] \le A[i{:}j]$, $i \le z \le j$
    end ,
```

and assume that we wish instead to

achieve $A[z] \le A[i{:}j]$, $i \le z \le j$, $x = A[z]$ **varying** z, x .

The above program only achieves the first two conjuncts of the new goal; we must extend the program to achieve the additional conjunct $x = A[z]$ as well.

One simple way to accomplish this would be to split the new goal into two disjoint goals:

```
achieve  $A[z] \le A[i:j]$,  $i \le z \le j$  varying  $z$
achieve  $x = A[z]$  varying  $x$ .
```

For the first, we already have a program; for the second, we may simply append the assignment

$$x := A[z] \ .$$

A second possibility would be to begin by achieving the relation $x = A[z]$, and then protect that relation while achieving $A[z] \le A[i:j]$ via P_2. In other words the relation $x = A[z]$ should be a global invariant of P_2, holding throughout execution of the program. In order to accomplish this goal, viz.

```
achieve  $x = A[z]$  in $P_2$  varying  $x$ ,
```

we must set x to the appropriate value whenever the variable z changes value. The assignments to z are

$$z := j \qquad z := y \ .$$

When z is initialized to j, we initialize $x = A[z]$ to $A[j]$; when z is reset to y, we reset x to $A[y]$. This yields the program

```
(z, y, x) := (j, j, A[j])
loop assert  $A[z] \le A[y:j]$,  $y \le z \le j$
     until  $y = i$
     $y := y - 1$
     if  $A[y] < A[z]$  then  $(z, x) := (y, A[y])$  fi
     repeat .
```

As it stands now, the first alternative requires less computation. But since we have established the global invariant $x = A[z]$, the conditional test $A[y] < A[z]$ may be simplified to $A[y] < x$. The final version of this program is

```
assert x=A[z] in
P_2': begin comment extended array-minimum program
    assert i,j∈N, i≤j
    (z,y,x) := (j,j,A[j])
    loop assert A[z]≤A[y:j], y≤z≤j
        until y=i
        y := y-1
        if A[y]<x then (z,x) := (y,A[y]) fi
        repeat
    assert A[z]≤A[i:j], i≤z≤j
    end .
```

In a similar manner, we could begin with the program

```
P_2'': begin comment extended array-minimum program
    assert i,j∈N, i≤j
    (y,x) := (j,A[j])
    loop assert x≤A[y:j]
        until y=i
        y := y-1
        if A[y]<x then x := A[y] fi
        repeat
    assert x≤A[i:j]
    end
```

that only achieves $x \le A[i:j]$, and extend it to achieve $x = A[z]$ as well. There is no easy way to set z at the end of P_2'' so that $x = A[z]$. But we can

achieve $x = A[z]$ **in** P_2'' **varying** z

by examining the assignments to x,

$x := A[j] \qquad x := A[y]$.

The corresponding assignments to z would be

$z := j \qquad z := y$,

yielding the same program as P_2'.

Example 2: *Integer Square-root.*

In our program,

```
P_1': begin comment integer square-root program
    assert $a \in N$
    $z := 0$
    loop assert $z^2 \le a,\ z \in N$
        until $a < (z+1)^2$
        $z := z+1$
        repeat
    assert $z = \lfloor \sqrt{a} \rfloor$
    end ,
```

the exit test $a<(z+1)^2$ is relatively difficult to compute as it involves squaring. It may be replaced by $a<s$, if we can extend P_1 to achieve the global invariant $s=(z+1)^2$:

achieve $s=(z+1)^2$ **in** P_1 **varying** s .

The variable z is set by two assignments in the program

$z := 0 \qquad z := z+1$;

we must assign appropriate values to s to maintain the desired relation $s=(z+1)^2$.

One way to accomplish this is to use a collection of rules that relate assignment statements to the values of the program variables. One of them states that in order for a global assertion of the form

assert $(y-b_0)\cdot 2\cdot a_2{}^2 = (x-a_0)\cdot[b_1\cdot(x-a_0-a_2)+2\cdot a_2\cdot(b_2+b_1\cdot a_0)]$ **in** P

holds if the assignments to x and y in P are of one of the two forms

$(x,y) := (a_0, b_0)$

or

$(x,y) := (x+a_2, y+b_1\cdot x+b_2)$,

where a_0 , b_0 , a_2 , b_1 , and b_2 are of constant value in P .

Let us try to apply this rule to the problem at hand. We match x and y in the rule with z and s in the program, respectively. The assignments to z are

$z := 0 \qquad z := z+1$,

so we let $a_0 \Rightarrow 0$ and $a_2 \Rightarrow 1$. Thus, assignments of the form

$$(z,s) := (0,b_0) \qquad (z,s) := (z+1, s+b_1\cdot z+b_2)$$

achieve the relation

$$(s-b_0)\cdot 2\cdot 1^2=(z-0)\cdot[b_1\cdot(z-0-1)+2\cdot 1\cdot(b_2+b_1\cdot 0)]$$

i.e.

$$(s-b_0)\cdot 2=z\cdot[b_1\cdot(z-1)+2\cdot b_2] \ .$$

So we are looking for instantiations of b_0 , b_1 , and b_2 , such that

$$(s-b_0)\cdot 2=z\cdot[b_1\cdot(z-1)+2\cdot b_2] \Rightarrow s=(z+1)^2 \ .$$

Isolating s to the left of the equality and matching, leaves

$$z\cdot[b_1\cdot(z-1)/2+b_2]+b_0 \Rightarrow (z+1)^2 \ .$$

Transforming $(z+1)^2$ into $z\cdot(z+2)+1$, suggests $b_0 \Rightarrow 1$ and

$$b_1\cdot(z-1)/2+b_2 \Rightarrow z+2 \ ,$$

which, in turn, suggests $b_1 \Rightarrow 2$ and $b_2 \Rightarrow 3$. Instantiating b_0 , b_1 , and b_2 back into the assignments, we get

$$(z,s) := (0,1) \qquad (z,s) := (z+1, s+2\cdot z+3)$$

A further improvement would be to

achieve $t=2\cdot z+3$ **in** P_1' .

By another rule (<> in the appendix), to get

assert $a_1\cdot(y-b_0)=b_1\cdot(x-a_0)$ **in** P ,

the proper assignments are

$$(x,y) := (a_0,b_0) \quad (x,y) := (x+a_1\cdot u, y+b_1\cdot u) \ .$$

This suggests the instantiation

$$(x \Rightarrow z, y \Rightarrow t, a_0 \Rightarrow 0, a_1 \Rightarrow 1, u \Rightarrow 1) \ ,$$

leaving

$$(t-b_0)=b_1\cdot z \Rightarrow t=2\cdot z+3 \ .$$

Taking $b_0 \Rightarrow 3$ and $b_1 \Rightarrow 2$, we get the assignments

$$(z,s,t) := (0,1,3) \qquad (z,s,t) := (z+1, s+t, t+2) ,$$

and the program

```
assert  $s=(z+1)^2$,  $t=2\cdot z+3$  in
$P_1''$: begin  comment famous integer square-root program
    assert  $a \in \mathbb{N}$
    $(z,s,t)$ := $(0,1,3)$
    loop assert  $z^2 \le a$,  $z \in \mathbb{N}$
        until  $a < s$
        $(z,s,t)$ := $(z+1, s+t, t+2)$
        repeat
    assert  $z = \lfloor \sqrt{a} \rfloor$
    end .
```

Example 3: *Binary Integer Square-root.*

At the conclusion of the overview chapter, we had obtained the following binary integer square-root program:

```
assert  $a \in \mathbb{N}$,  $z \in \mathbb{N}$,  $y \in 2^{\mathbb{N}}$  in
$P_5$: begin  comment binary integer square-root program
    assert  $a \in \mathbb{N}$
    $(z,y)$ := $(0,1)$
    loop until  $a < y^2$
        $y$ := $2\cdot y$
        repeat
    loop assert  $z \le \sqrt{a}$,  $\sqrt{a} < z+y$
        until  $y \le 1$
        $y$ := $y/2$
        if  $(z+y)^2 \le a$  then  $z$ := $z+y$  fi
        repeat
    assert  $z \le \sqrt{a}$,  $\sqrt{a} < z+1$,  $z \in \mathbb{N}$
    end .
```

In this program, the exit test $a<y^2$ and conditional test $(z+y)^2 \le a$ are the most expensive expressions to compute. The latter is equivalent to $z^2+2\cdot y\cdot z+y^2 \le a$, and assuming that we can

achieve $u=z^2,\ v=2\cdot y\cdot z,\ w=y^2$ **in** P, **varying** u, v, w ,

we may replace the tests with $a<w$ and $u+v+w \le a$, respectively.

Whenever z or y is updated, the new variables u, v, and w must be updated correspondingly so that their relations with z and y remain invariant. In a manner similar to the previous example, we obtain

```
assert $a \in \mathbb{N},\ z \in \mathbb{N},\ y \in 2^{\mathbb{N}},\ u=z^2,\ v=2\cdot y\cdot z,\ w=y^2$ in
$P_5$: begin comment extended binary integer square-root program
    assert $a \in \mathbb{N}$
    $(z, y, u, v, w) := (0, 1, 0, 0, 1)$
    loop until $a<w$
        $(y, w) := (2\cdot y, 4\cdot w)$
        repeat
    loop assert $z \le \sqrt{a},\ \sqrt{a} < z+y$
        until $y \le 1$
        $(y, v, w) := (y/2, v/2, w/4)$
        if $u+v+w \le a$ then $(z, u, v) := (z+y, u+v+w, v+2\cdot w)$ fi
        repeat
    assert $z \le \sqrt{a},\ \sqrt{a} < z+1,\ z \in \mathbb{N}$
    end .
```

The variable z affects only the value of z itself, so it is a candidate for elimination from the program. The only problem is that it is z that contains the desired final result. Fortunately, since we have the global invariant $v=2\cdot y\cdot z$, we can just append the goal

achieve $v=2\cdot y\cdot z$ **varying** z

to the end of the program. We shall return to this unachieved goal later.

Once we have eliminated the assignment $z:=z+y$, the variable y only affects the the exit test $y \le 1$. But we can replace the variable y with $\sqrt{w}$, since $w=y^2$ and y is known to be positive. This gives us the exit clause

until $\sqrt{w} \le 1$,

for which the primitive

```
    until $w \le 1$
```

may be substituted.

Squeezing the last drop of ink out of this example, we obtain yet another slight improvement: To transform the test $u+v+w \le a \Rightarrow u+v+w \le 0$, we can apply the transformation $u \Rightarrow u+a$, yielding the initialization

$$(u, v, w) := (-a, 0, 1)$$

and conditional

```
    if $u+v+w \le 0$ then $(u,v) := (u+v+w, v+2\cdot w)$ fi .
```

To avoid recomputing the expression $u+v+w$ for both the conditional test $u+v+w \le a$ and the assignment $u:=u+v+w$, a temporary variable could be generated, say t, such that $t=u+v+w$. It would then be used in the test $t \le 0$ and assignment $(u,v):=(t, v+2\cdot w)$.

Incorporating the above improvements, we obtain

```
assert $a \in N$,  $z \in N$,  $y \in 2^N$,  $u+a=z^2$,  $v=2\cdot y\cdot z$,  $w=y^2$  in
$P_3'$: begin comment optimized integer square-root program
    assert $a \in N$
    $(u,v,w) := (-a,0,1)$
    loop until $a<w$
        $w := 4\cdot w$
        repeat
    loop assert $z \le \sqrt{a}$,  $\sqrt{a}<z+y$
        until $w \le 1$
        $(v,w) := (v/2, w/4)$
        $t := u+v+w$
        if $t \le 0$ then $(u,v) := (t, v+2\cdot w)$ fi
        repeat
    achieve $v=2\cdot y\cdot z$ varying $z$
    assert $z \le \sqrt{a}$,  $\sqrt{a}<z+1$,  $z \in N$
    end .
```

The variables z and y have been left in the assertions though they have been eliminated from the program text; at any point in which these "ghost" variables are undefined by the program, the intent of the assertions is that there exist values for z and y such that the assertions hold.

Finally, another drop of computation is saved by

achieve $s=v+w$ **varying** s .

Since $v=0$ upon entering the second loop, we set $s:=w$ at that point. When v is halved and w is quartered, the new value of s should become $v/2+w/4=v/2+w/2-w/4=s/2-w/4$. So as not to compute $w/4$ twice, we may first assign $w:=w/4$ and then $s:=s/2-w$. When v is incremented by $2\cdot w$, so must s be. We can now eliminate v from the program. For the second loop, we have

```
s := w
loop assert $z \le \sqrt{a}$, $\sqrt{a} < z+y$
     until $w \le 1$
     w := w/4
     s := s/2-w
     t := u+s
     if $t \le 0$ then (u,s) := (t,s+2·w) fi
     repeat .
```

We still have the goal

achieve $v=2\cdot y\cdot z$ **varying** z .

Though y and v are now ghost variables, we know that $s=v+w$, $w=y^2$, and $y\epsilon 2^N$. From these global invariants and the exit test $w\le 1$, we conclude that $w=y=1$ upon termination of the second loop. Thus, to achieve the above goal, we need only

achieve $s-1=2\cdot 1\cdot z$ **varying** z ;

we therefore assign

$z := (s-1)/2$.

The transformation $w \Rightarrow w/2$ gives us our final version of this program (cf. Dijkstra [1976]):

```
assert $a \in N$, $z \in N$, $y \in 2^N$, $u+a=z^2$, $v=2 \cdot y \cdot z$, $w=2 \cdot y^2$, $s=v+w/2$ in
$P_s''$: begin comment improved integer square-root program
    assert $a \in N$
    $(u,w)$ := $(-a,2)$
    loop until $2 \cdot a < w$
        $w$ := $4 \cdot w$
        repeat
    $s$ := $w/2$
    loop assert $z \le \sqrt{a}$, $\sqrt{a} < z+y$
        until $w \le 2$
        $w$ := $w/4$
        $s$ := $(s-w)/2$
        $t$ := $u+s$
        if $t \le 0$ then $(u,s)$ := $(t, s+w)$ fi
        repeat
    $z$ := $(s-1)/2$
    assert $z \le \sqrt{a}$, $\sqrt{a} < z+1$, $z \in N$
    end .
```

We have successfully replaced the original exit test $a<y^2$ and conditional test $(z+y)^2 \le a$, both of which involve squaring, a relatively expensive operation, with the simple tests $2 \cdot a < w$ and $t \le 0$. This, at the cost of updating the variables by addition/subtraction and multiplication/division by powers of two, relatively cheap operations on binary computers.

The real square-root program

```
begin comment real square-root program
    assert $0 \le a < 1$, $0 < \epsilon$
    $(z,y)$ := $(0,1)$
    loop assert $z \le \sqrt{a}$, $\sqrt{a} < z+y$
        until $y \le \epsilon$
        $y$ := $y/2$
        if $(z+y)^2 \le a$ then $z$ := $z+y$ fi
        repeat
    assert $z \le \sqrt{a}$, $\sqrt{a} < z+\epsilon$
    end
```

may be optimized in a similar manner to obtain Wensley's [1959] square-root algorithm:

```
assert $a+4\cdot u\cdot y=z^2$ in
begin comment optimized real square-root program
    assert $0\le a<1,\ 0<e$
    $(z,y,u)$ := $(0, 1/2, -a/2)$
    loop assert $z\le\sqrt{a},\ \sqrt{a}<z+2\cdot y$
        until $y\le 2\cdot e$
        $(y,u)$ := $(y/2, 2\cdot u)$
        $t$ := $u+z+y$
        if $t\le 0$ then $(z,u)$ := $(z+2\cdot y, t)$ fi
        repeat
    assert $z\le\sqrt{a},\ \sqrt{a}<z+e$
    end .
```

In this chapter, we have seen how to systematically develop a program from its specifications; in the next chapter we shall see how similar ideas may be employed to generate the invariants from the code.

CHAPTER VI

PROGRAM ANNOTATION

1. INTRODUCTION

As we have seen, invariant assertions are often needed for modifications to carry through. But what if the programmer failed to supply enough of them? In particular, if the program is incorrect with respect to its specifications, then, perforce, some of the given assertions (at very least, the output specification) do not reflect what the program is actually doing. And without knowing what the program is doing, we cannot proceed to debug it.

Program annotation is the process of discovering invariant assertions from the program text itself. Our task is to generate the invariants describing the workings of the program as is, independent of its correctness or incorrectness. The process is iterative, since finding some invariants suggests others. Assertions supplied by the programmer cannot be assumed true, though they may be used to guide the search for correct invariants.

If the invariants associated with the point of termination of a program imply that the given output specification is true for any input satisfying the input specification, then the program has been proved *correct*. On the other hand, if there exist legal input values such that whenever the output invariants hold for those input values, the specifications do not all hold, then the program is *incorrect*. In this manner, invariants are used for proving correctness/incorrectness of programs.

Existing implementations of the invariant-assertion method of program verification are not fully mechanical; the user must supply most, if not all, of the invariants himself. If the original program is not supplied with sufficient invariants to prove correctness or incorrectness, they must be supplemented. These invariants then enable one to verify if what the program does is what it was intended to do. Invariants are also useful in analyzing other properties of programs, e.g. time complexity.

In the following sections, we present a unified approach to program annotation, using *annotation rules* — in the style of Hoare [1969] — to derive invariants. Section 2 is an overview. It is followed by two detailed examples: the first illustrates the basic techniques on a single-loop program; the second applies the techniques to a program with nested loops and arrays.

Three earlier annotation systems are:

- the system described in Elspas [1974], based mainly upon the solution of difference equations;
- VISTA (German [1974], German and Wegbreit [1975]), based upon the top-down heuristics of Wegbreit [1974]; and
- ADI (Tamir [1976]), an interactive system based upon the methods of Katz and Manna [1976] and Katz [1976].

Our annotation system, as described here, attempts to incorporate and expand upon those systems. Recently, Suzuki and Ishihata [1977] and German [1978] have implemented systems that generate invariants useful in checking for various runtime errors.

2. OVERVIEW

In this section, we first define some terminology and then present samples of each type of annotation rule.

1. *Notation and Terminology*

Given a program with its specifications, our goal is to document the program automatically with invariants. If the program is correct with respect to the specifications, we would like the invariants to provide sufficient information to demonstrate its correctness; if the program is incorrect, we would like information helpful in determining what is wrong with it.

We shall be dealing with three types of assertions:

● *Global invariants* are relations that hold at all places (i.e. labels) and at all times during the execution of some program segment. We write

assert α **in** P

to indicate that the relation α is a global invariant in a program segment P . (Actually, α is considered a global invariant even if it only begins to hold once the variables in α have been assigned an initial value within P .)

● *Local invariants* are associated with specific points in the program, and hold for the current values of the variables whenever control passes through the corresponding point. Thus,

L: **assert** α

means that the relation α holds each time control is at label L .

● *Candidate assertions*, also associated with specific points, are relations hypothesized to be local invariants, but that have not yet been verified. We write

L: **suggest** α .

Consider the following simple program, meant to compute the quotient q and remainder r of the integer input values c and d :

```
P_0: begin comment integer-division program
    B_0: assert c∈N, d∈N+1
    (q,r) := (0,c)
    loop L_0: assert . . .
        until r<d
        (q,r) := (q+1,r-d)
        repeat
    E_0: suggest q≤c/d, c/d<q+1, q∈Z, r=c-q·d
    end ,
```

where N is the set of nonnegative integers, N+1 is the set of positive integers, and Z is the set of all integers. This program will be used only to illustrate various aspects of program annotation; complete examples of annotation are given in the next section.

The invariant

assert $c \in \mathbb{N},\ d \in \mathbb{N}+1$

attached to the **begin**-label B_0 is the input specification of the program defining the class of "legal" inputs. The input specification is assumed to hold, regardless of whether the program is correct or not.

The candidate

suggest $q \leq c/d,\ c/d < q+1,\ q \in \mathbb{Z},\ r = c - q \cdot d$

attached to the **end**-label E_0 is the output specification of the program. It states that the desired outcome of the program is that q be the largest integer not larger than c/d and r be the remainder. Since one cannot assume that the programmer has not erred, initially all programmer-supplied assertions — including the program's output specification — are only candidates for invariants.

In order to verify that a candidate is indeed a local invariant, we must show that whenever control reaches the corresponding point, the candidate holds. Suppose that we are given a candidate for a loop invariant

L_0: **suggest** $r = c - q \cdot d$.

To prove that it is an invariant, one must show: 1) that the relation holds at L_0 when the loop is first entered, and 2) that once it holds at L_0, it remains true each subsequent time control returns to L_0. If we succeed, then we would write

L_0: **assert** $r=c-q \cdot d$.

Furthermore, if $r=c-q \cdot d$ holds whenever control is at L_0, then it will also hold whenever control leaves the loop and reaches E_0. In other words, $r=c-q \cdot d$ would also be an invariant at E_0 and may be removed from the list of candidates at E_0. In that case, we would write

E_0: **assert** $r=c-q \cdot d$ **and suggest** $q \le c/d,\ c/d < q+1,\ q \in \mathbb{Z}$.

Global invariants often express the range of variables. For example, since the variable q is first initialized to 0 and then repeatedly incremented by 1, it is obvious that the value of q is always a nonnegative integer. Thus we have the global invariant

assert $q \in \mathbb{N}$ **in** P_0

that relates to the program as a whole and states that $q \in \mathbb{N}$ throughout execution of the program segment P_0.

In this chapter, we describe various annotation techniques. These techniques are expressed as rules: the antecedents of each rule are usually annotated program segments containing invariants or candidate invariants and the consequent is either an invariant or a candidate. This list is representative of the kinds of rules that may be used for annotation; it is not, however, meant to be a complete list. Not only are these rules useful for automatic or interactive annotation, but they help to clarify the interrelation between program text and invariants for the programmer.

We differentiate between three types of rules: assignment rules, control rules, and heuristic rules.

- *Assignment rules* yield *global* invariants based only upon the assignment statements of the program.
- *Control rules* yield *local* invariants based upon the control structure of the program.
- *Heuristic rules* have *candidates* as their consequents. These candidates, though promising, are not guaranteed to be invariants.

The assignment and control rules are *algorithmic* in the sense that they derive relations in such a manner as to *guarantee* that they are invariants. The heuristics are rules of *plausible* inference, reflecting common programming practice.

2. *Assignment Rules*

Many of the algorithmic rules depend only upon the assignment statements of the program and not upon its control structure. In other words, whether the assignments appear within an iterative or recursive loop or on some branch of a conditional statement is irrelevant. Since the location and order in which assignments are executed does not affect the validity of the rules, these rules yield global invariants.

The various assignment rules relate to particular operators occurring in the assignment statements of the program. Some of the rules for addition, for example, are: an *addition rule* that gives the range of a variable that is updated by adding (or subtracting) a constant; a *set-addition rule* for the case where the variable is added to another variable whose range is already known; and an *addition-relation rule* that relates two variables that are always incremented by similar expressions. Corresponding rules apply to other operators.

For example, the *addition rule* is

$$\frac{x := a_0 \quad x := x+a_1 \quad x := x+a_2 \quad \ldots \quad \textbf{in } P}{\textbf{assert } \; x \in a_0+a_1 \cdot \mathrm{N}+a_2 \cdot \mathrm{N}+ \ldots \quad \textbf{in } P} ,$$

where P is a program segment and the expressions a_i are of constant value within P. The antecedent

$$x := a_0 \quad x := x+a_1 \quad x := x+a_2 \quad \ldots \quad \textbf{in } P$$

indicates that the *only* assignments to the variable x in P are $x{:=}a_0$, $x{:=}x+a_1$, $x{:=}x+a_2$, etc. The consequent

$$\textbf{assert } \; x \in a_0+a_1 \cdot \mathrm{N}+a_2 \cdot \mathrm{N}+ \ldots \quad \textbf{in } P$$

is a global invariant indicating that x belongs to the set $a_0+a_1 \cdot \mathrm{N}+a_2 \cdot \mathrm{N}+ \ldots$ throughout execution of P — but *only* from the point when x first receives a defined value in P via the assignment $x{:=}a_0$. (After any execution of $x{:=}a_0$, clearly $x \in a_0+a_1 \cdot \mathrm{N}+a_2 \cdot \mathrm{N}+ \ldots$ with $x=a_0+a_1 \cdot 0+a_2 \cdot 0+ \ldots$, and if $x=a_0+a_1 \cdot m+a_2 \cdot n+ \ldots$ for some m, $n, \ldots$ before executing $x{:=}x+a_1$, then $x=a_0+a_1 \cdot (m+1)+a_2 \cdot n+ \ldots$ after executing the assignment. Thus, m represents the number of executions of $x{:=}x+a_1$ since $x{:=}a_0$ was executed last, n is the number of executions of $x{:=}x+a_2$, etc.) From such an invariant, more specific properties may be derived. For example a bound on x may be derived using methods of *interval arithmetic* (see for example Gibb [1961]). Note that no restrictions are placed on the order in which the assignments to x are executed, except that *prior* to the *first* execution of $x{:=}a_0$ the invariant may not hold.

In our simple program P_0, the assignments to the variable q are

$$q{:=}0 \qquad q{:=}q+1 \ .$$

So we can apply the *addition rule*, instantiating a_0 with 0 and a_1 with 1, and obtain the global invariant $q \in 0+1\cdot\mathbb{N}$, i.e.

$$\textbf{assert } q\in\mathbb{N} \textbf{ in } P_0 \ .$$

The assignments to r in P_0 are

$$r{:=}c \qquad r{:=}r-d \ .$$

Applying the same rule to them, letting $a_0{=}c$ and $a_1{=}-d$, yields the invariant

$$\textbf{assert } r\in c-d\cdot\mathbb{N} \textbf{ in } P_0 \ .$$

Given that d is positive, we may conclude that $r\leq c$.

The *set-addition rule* is a more general form of the above *addition rule*, applicable to nondeterministic assignments of the form $x{:\in}f(S)$, where an arbitrary element in the set $f(S)=\{f(s): s\in S\}$ is assigned to x. Note that an assignment $x{:=}f(s)$, where it is only known that $s\in S$, may be viewed as the nondeterministic assignment $x{:\in}f(S)$. The *set-addition rule* is

$$\frac{x{:\in}S_0 \quad x{:\in}x+S_1 \quad x{:\in}x+S_2 \quad \ldots \quad \textbf{in } P}{\textbf{assert } x\in S_0+\Sigma S_1+\Sigma S_2+\ldots \quad \textbf{in } P} \ ,$$

where ΣS denotes the set of finite sums $s_1+s_2+\ldots+s_m$ for (not necessarily distinct) addends s_i in S. If $m{=}0$, the sum is 0; if S contains the single element s, then $\Sigma S{=}s\cdot\mathbb{N}$. (This rule applies analogously to any associative and commutative operator "$\oplus$".) These assignment rules for global invariants are related to the weak interpretation method of Sintzoff [1972] (see also Wegbreit [1975], Wegbreit and Spitzen [1976], and Harrison [1977]) that has been implemented by Scherlis [1974] and German and Wegbreit [1975].

In program P_0 the assignments to r were

$$r:=c \qquad r:=r-d \ .$$

Since we are given that $c \in N$ and $d \in N+1$, we may view these as the nondeterministic assignments

$$r:\in N \qquad r:\in r-(N+1) \ ,$$

and by applying the *set-addition rule* we obtain the global invariant $r \in N-\Sigma(N+1)$. This simplifies to

assert $r \in Z$ **in** P_0 ,

where Z is the set of all integers.

To relate different variables appearing in a program, we have an *addition-relation rule*:

$$\frac{\begin{array}{l}(x,y) := (a_0,b_0) \quad (x,y) := (x+a_1 \cdot u, y+b_1 \cdot u) \\ \qquad\qquad\qquad\qquad (x,y) := (x+a_1 \cdot v, y+b_1 \cdot v) \quad \ldots \quad \textbf{in } P\end{array}}{\textbf{assert } a_1 \cdot (y-b_0)=b_1 \cdot (x-a_0) \textbf{ in } P} \ ,$$

where u, v, ..., are arbitrary (not necessarily constant) expressions. The invariant begins to hold only when the multiple assignment $(x,y):=(a_0,b_0)$ has been executed for the first time. (The invariant $a_1 \cdot (y-b_0)=b_1 \cdot (x-a_0)$ clearly holds when $x=a_0$ and $y=b_0$. Assuming it holds before executing $(x,y):=(x+a_1 \cdot u, y+b_1 \cdot u)$, then after executing the assignment, both sides of the equality are increased by $a_1 \cdot b_1 \cdot u$, and the invariant still holds.) The multiple assignments in the antecedent of such rules, e.g. $(x,y):=(x+a_1 \cdot u, y+b_1 \cdot u)$, may represent the cumulative effect of individual assignments lying on a path between two labels, with the understanding that whenever $x:=x+a_1 \cdot u$ is executed, so is $y:=y+b_1 \cdot u$ for the same value of the expression u. In that case, the invariant will not, in general, hold between the individual assignments.

In our example, the assignments in the initialization path give us

$$(q,r):=(0,c) \ ,$$

and for the loop-body path we have

$$(q,r):=(q+1,r-d) \ .$$

By a simple application of the *addition-relation rule* with $a_0=0$, $b_0=c$, $a_1=u=v=1$, and $b_1=-d$, we derive the invariant $1 \cdot (r-c)=-d \cdot (q-0)$, which simplifies to

$$\textbf{assert}\ r=c-q\cdot d\ \ \textbf{in}\ P_0\ .$$

Note that this *addition-relation rule* (as well as several other relation rules) may be derived from the following general relation-rule schema:

$$\begin{array}{l}
(x,y) := (a_0,b_0) \quad (x,y) := (x\oplus(u\otimes a_1), y\oplus(u\otimes b_1)) \\
\qquad\qquad\qquad\qquad (x,y) := (x\oplus(v\otimes a_1), y\oplus(v\otimes b_1)) \quad \ldots \quad \textbf{in}\ P \\
\hline
\textbf{assert}\ \ (a_0\otimes b_1)\oplus(y\otimes a_1)=(b_0\otimes a_1)\oplus(x\otimes b_1) \quad in\ P\ ,
\end{array}$$

where the operator $\oplus$ is commutative and associative, operator $\otimes$ satisfies $(a\otimes b)\otimes c=(a\otimes c)\otimes b$, and $(a\oplus b)\otimes c=(a\otimes c)\oplus(b\otimes c)$. The various relation rules are related to optimization and extension techniques, where the desired relation is given along with the assignments to one of the variables, and the proper assignment to the other variable is sought (as in the previous chapter). For related approaches see Caplain [1975] and German [1978].

3. *Control Rules*

Unlike the previous rules that completely ignore the control structure of the program, *control rules* derive important invariants from the program structure; they are directly related to the verification rules of Hoare [1969]. There are, for example, two rules to push invariants forward in a loop. The *forward loop-exit rule*,

$$\begin{array}{l}
\textbf{loop}\ P' \\
\qquad \textbf{assert}\ \alpha \\
\qquad \textbf{until}\ t \\
\qquad L': \\
\qquad P'' \\
\qquad \textbf{repeat} \\
L'': \\
\hline
L':\ \textbf{assert}\ \ \alpha,\ \neg t \\
L'':\ \textbf{assert}\ \ \alpha,\ t
\end{array}$$

reflects the fact that if execution of a loop terminates at L'' , then the exit test t must have just held, while if the loop is continued at L' , the exit test was false. Furthermore, any relation α that held just prior to the test, also holds immediately after. The *forward loop-body rule*,

```
assert $\alpha$
loop $L$:
     $P$
     assert $\beta$
     repeat
─────────────────
$L$: assert $\alpha \vee \beta$
```

states that for control to be at the head of a loop, at L, either the loop has just been entered, or the loop body has been executed and the loop is being repeated. Therefore the disjunction $\alpha \vee \beta$ of an invariant α known to hold just before the loop and an invariant β known to hold at the end of the loop body must hold at L.

Applying the first rule to the loop in the integer-division program P_0 yields the invariant $r<d$ at E_0 and $r \geq d$ at the head of the loop body:

```
$(q,r)$ := $(0,c)$
loop $L_0$:
     until $r<d$
     assert $r \geq d$
     $(q,r)$ := $(q+1, r-d)$
     repeat
$E_0$: assert $r<d$ .
```

To propagate invariants, such as $r \geq d$, past assignment statements, we have a *forward assignment rule,*

```
assert $\alpha(x,y)$
$x$ := $f(x,y)$
$L$:
─────────────────
$L$: assert $\alpha(f^-(x,y),y)$ ,
```

where f^- is the inverse (assuming that there is an inverse) of the function f in the first argument, i.e. $f^-(f(x,y),y)=x$. By using the inverse function f^-, the value of x prior to the assignment may be expressed in terms of the current value of x as $f^-(x,y)$. Thus, if the relation $\alpha(x,y)$ held before the assignment to x, then after the assignment $\alpha(f^-(x,y),y)$ holds. Even if there is no inverse function, variants of this rule may often be used to glean some useful information.

In our example, since the first loop-body assignment $q:=q+1$ does not affect any variable appearing in the invariant $r \geq d$, the invariant is pushed forward unchanged. To propagate $r \geq d$ past the second assignment, $r:=r-d$, we replace r by the inverse of $r-d$, that is $r+d$, yielding $r+d \geq d$, or

```
assert $r \geq 0$ ,
```

at the end of the loop body.

We have an *assignment axiom*

```
$x$ := $a$
assert $x=a$ ,
```

where the expression a must remain constant during execution of the assignment; in other words, it may not contain x. This axiom gives us the invariant

```
assert $r=c$
```

prior to entering the loop. Thus, by the second rule for loops, the *forward loop-body*, we get the loop invariant

```
$L_0$: assert $r=c \vee r \geq 0$ .
```

Since, by the input specification $0 \leq c$, the first disjunct implies the second, this invariant simplifies to

```
$L_0$: assert $r \geq 0$ .
```

To generate invariants from a conditional test, we have a *forward test rule*:

```
assert $\alpha$
if $t$ then  $L'$:
             $P'$
     else    $L''$:
             $P''$
     fi
──────────────────
$L'$: assert $\alpha$, $t$
$L''$: assert $\alpha$, $\neg t$ .
```

That is, for the **then**-branch to be taken t must be true, while for the **else**-branch to be taken it must be false; furthermore, any α that held before the test, also holds after. Once invariants have been generated for the two branches, they are pushed forward by the *forward branch rule*:

```
if t then  P'
           assert α
     else  P''
           assert β
     fi
L:
________________________
L: assert α∨β .
```

It states that for control to be at the point after the conditional statement, one of the two branches must have been traversed.

The following *forall rule* is valuable for programs with universally-quantified output specification. Given a loop invariant $\alpha(x)$ at L containing the integer variable (or expression) x and *no* other variables, check if x is monotonically increasing by one. If it is, then we have as a loop invariant at L that α still holds for all intermediate values lying between the initial and current values. That is

```
assert x=a, x∈Z
loop L: assert α(x)
     P
     assert x=x_L+1
     repeat
__________________________
L: assert (∀ζ∈Z)(a≤ζ≤x)α(ζ) ,
```

where a is an integer expression with a constant value in P and x_L is the value of x when last at L. (This rule is similar to the universal-quantification technique for arrays in Katz and Manna [1973].) The *forall rule* may be broadened to apply when x is increasing by an amount other than 1, or for a decreasing x.

4. *Schematic Rules*

In this subsection, we shall illustrate how the control rules may be used to derive annotation rules for program schemata.

Consider, for example, the following single-loop, single-conditional, program schema:

```
P*: begin comment single-loop schema
    z := c
    loop L*: assert ...
         until t(z)
         z := f(z)
         if s(z) then z := g(z) else z := h(z) fi
         repeat
    end .
```

We shall assume that the inverse functions f^-, g^-, and h^- are available whenever required by the rules.

The *assignment axiom*, applied to the initial assignment $z{:=}c$ yields the invariant

```
assert z=c
```

before the loop. The *forward loop-exit rule* generates the invariant $\neg t(z)$ at the head of the loop body, immediately after the **until**-clause, and then the *forward assignment rule* gives $\neg t(f^-(z))$ preceding the conditional.:

```
assert ¬t(f⁻(z))
if s(z) then z := g(z) else z := h(z) fi .
```

The *forward test rule* propagates that invariant forward, adding $s(z)$ at the head of the **then**-clause, and $\neg s(z)$ at the head of the **else**-clause:

```
if s(z)   then assert ¬t(f⁻(z)), s(z)
               z := g(z)
          else assert ¬t(f⁻(z)), ¬s(z)
               z := h(z)
          fi .
```

By pushing $\neg t(f^-(z))$ and $s(z)$ through the **then**-branch assignment $z{:=}g(z)$, and $\neg t(f^-(z))$ and $\neg s(z)$ through the **else**-branch assignment $z{:=}h(z)$, we get

```
if s(z)   then z := g(z)
               assert ¬t(f⁻(g⁻(z))), s(g⁻(z))
          else z := h(z)
               assert ¬t(f⁻(h⁻(z))), ¬s(h⁻(z))
          fi .
```

Combining the invariants from the two different paths — using the *forward branch rule* — one gets

$$\textbf{assert}\ [\neg t(f^{-}(g^{-}(z))) \wedge s(g^{-}(z))] \vee [\neg t(f^{-}(h^{-}(z))) \wedge \neg s(h^{-}(z))]$$

after the conditional, at the end of the loop body.

The *forward loop-body rule* expressed the fact that if control is at the head of a loop, either the loop-initialization invariant or the loop-body invariant must hold. Applying this rule to our schema

```
assert $z=c$
loop $L^*$: assert . . .
     until $t(z)$
     $z := f(z)$
     if $s(z)$ then $z := g(z)$ else $z := h(z)$ fi
     assert $[\neg t(f^{-}(g^{-}(z))) \wedge s(g^{-}(z))] \vee [\neg t(f^{-}(h^{-}(z))) \wedge \neg s(h^{-}(z))]$
     repeat
```

we derive the loop invariant

$$L^*\text{: } \textbf{assert}\ z=c \vee [\neg t(f^{-}(g^{-}(z)))\wedge s(g^{-}(z))] \vee [\neg t(f^{-}(h^{-}(z)))\wedge\neg s(h^{-}(z))]$$

This loop invariant embodies two facts about the control structure of this schema:

● Whenever control is at L^*, either the loop has just been entered, or the loop-exit test was false the last time around the loop. That is,

$$L^*\text{: } \textbf{assert}\ z=c \vee \neg t(f^{-}(g^{-}(z))) \vee \neg t(f^{-}(h^{-}(z))) .$$

The first disjunct is the result of the initialization path; the second states that the exit test was false for the value of z when L^* was last visited, assuming control came via the **then**-path of the conditional; the third disjunct says the same for the case when control came via the **else**-path.

● Whenever control is at L^*, either the loop has just been entered, or the conditional test was true the last time around and the **then**-path was taken, or the test was false and the **else**-path was taken. That is,

$$L^*\text{: } \textbf{assert}\ z=c \vee s(g^{-}(z)) \vee \neg s(h^{-}(z)) .$$

As another simple example, consider the loop schema

```
z := 0
loop L:
     until  t(z)
     z := z+1
     repeat
E: .
```

By the *label axiom*

L: **assert** $x=x_L$,

we get

L: **assert** $z=z_L$

Thus, we can easily derive the following invariants:

```
z := 0
assert  $z=0$
loop L: assert  $z=z_L$
     until  t(z)
     z := z+1
     assert  $z-1=z_L,\ \neg t(z-1)$
     repeat
E: assert  t(z) .
```

Now, by the *forward loop-body rule* we can derive the invariant

L: **assert** $z=0 \vee \neg t(z-1)$

and by the *forall rule* , we get

L: **assert** $(\forall \zeta \in Z)(0 \le \zeta \le z)(\zeta=0 \vee \neg t(\zeta-1))$.

This simplifies to

L: **assert** $(\forall \zeta \in N)(\zeta < z) \neg t(\zeta)$;

combined with the invariant $t(z)$ that holds at E, it implies that the final value of z is the minimum nonnegative integer satisfying the predicate t:

```
E: assert  $z=(min\ \zeta\in N)t(\zeta)$ .
```

5. *Heuristic Rules*

In contrast with the above rules that are algorithmic in the sense that they derive relations that are guaranteed to be invariants, there is another class of rules, *heuristic rules*, that can only suggest candidates for invariants. These candidates must be verified.

As an example, consider the following *conditional heuristic*

```
if $t$ then  $P'$
             assert $\alpha$
     else    $P''$
             assert $\beta$
     fi
L:
________________________
L: suggest $\alpha$, $\beta$ .
```

Since we know that α holds if the **then**-path P' is taken, while β holds if the **else**-path P'' is taken, clearly their disjunction $\alpha \vee \beta$ holds at L in either case (that was expressed in the *forward branch rule*). However, since in constructing a program, a conditional statement is often used to achieve the same relation in alternative cases (cf. the *conditional* synthesis rule, page 93), it is plausible that α (or, by the same token, β) may hold true for *both* the **then**- and **else**-paths.

As mentioned earlier, the output specification and user-supplied assertions are the initial set of candidates. Candidates are propagated over assignment and conditional statements using the same control rules as for invariants. The *top-down heuristic*,

```
assert $\alpha$
loop L:
    until $t$
    $P$
    repeat
suggest $\gamma$

fact $\gamma$ when $\alpha$
________________________
L: suggest $\gamma$ ,
```

may be used to push a candidate (or invariant) γ backwards into a loop. Though $t \supset \gamma$ (i.e. $\neg t \vee \gamma$) would be a sufficiently strong loop invariant at L to establish γ upon loop exit, the heuristic suggests a stronger candidate, γ itself, at L. Since a necessary condition for γ to be an invariant is that it hold upon entrance to the loop, the second antecedent of the rule requires that the invariant α before the loop imply that γ holds. The idea underlying this heuristic is that an iterative loop is constructed in order to achieve a conjunctive goal by placing one conjunct of the goal in the exit test, and maintaining the other invariantly true (cf. the *forward loop* synthesis rule, page 97).

Wegbreit [1974] and Katz and Manna [1976] have suggested a more general form of these two heuristics:

$$\frac{L\text{: }\textbf{assert}\ \ \alpha \vee \beta}{L\text{: }\textbf{suggest}\ \ \alpha,\ \beta}\ .$$

However, as they remark, this heuristic should not be applied indiscriminately to any disjunctive invariant. We would not, for example, want to replace all occurrences of an invariant $x \geq 0$ with the candidates $x > 0$ and $x = 0$. Special cases, such as the above *conditional* and *top-down* heuristics are needed to indicate where the strategy is relatively likely to be profitable.

Returning to our integer-division example

```
P_0: begin. comment integer-division program
     B_0: assert c∈N, d∈N+1
     (q,r) := (0,c)
     loop L_0: assert  . . .
          until r<d
          (q,r) := (q+1,r-d)
          repeat
     E_0: suggest q≤c/d, c/d<q+1, q∈Z, r=c-q·d
     end ,
```

the *top-down heuristic* suggests that of the candidates

$$E_0\text{: }\textbf{suggest}\ \ q \leq c/d,\ c/d < q+1,\ q \in \mathbb{Z},\ r = c - q \cdot d\ ,$$

those that hold upon entering the loop — when $q=0$ and $r=c$ — are also candidates at L_0. They are

L_0: **suggest** $q \le c/d$, $q \in \mathbf{Z}$, $r = c - q \cdot d$.

The remaining candidate at E_0, $c/d < q+1$, does not necessarily hold for $q=0$.

Each candidate must be checked for invariance: it must hold for the loop-initialization path and must be maintained true around the loop. Of the three candidates at L_0, the last two, $q \in \mathbf{Z}$ and $r = c - q \cdot d$, have already been shown to be global invariants. To prove that the first, $q \le c/d$, is a loop invariant at L_0, we first try to show that it is true when the loop is entered, i.e. that

$$0 \le c/d \ .$$

The truth of this condition follows from the input specifications. Then we try to show that if $q \le c/d$ is true at L_0 and the loop is continued, then $q \le c/d$ holds when control returns to L_0, i.e.

$$q \le c/d \ \wedge \ r \ge d \quad \supset \quad q+1 \le c/d \ .$$

This condition, however, is not provable. Nevertheless, we can show that $q \le c/d$ is an invariant by making use of the global invariant $r = c - q \cdot d$. Substituting $c - q \cdot d$ for r in $r \ge d$ yields $c - q \cdot d \ge d$; it follows that the above implication holds and $q \le c/d$ is an invariant at L_0. Thus, while an attempt to directly verify the candidate $q \le c/d$ failed, once we have established that $r = c - q \cdot d$ is an invariant, we can also show that $q \le c/d$ is an invariant.

Indeed, in general there may be insufficient information to prove that a candidate is invariant when it is first suggested, and only when other invariants are subsequently discovered might it become possible to verify the candidate. Therefore, candidates should be retained until all invariants and candidates have been generated. Unproved candidates are also used by the heuristics to generate additional candidates. For example, the *top-down heuristic* uses the as yet unproved candidate γ to generate the loop candidate γ at L .

Another heuristic, valuable for loops with universally quantified output invariants, is the *generalization heuristic rule*

```
assert $x=a$
loop $L$: suggest $\alpha(x,y)$
     $P$
     assert $x=f(x_L)$
     repeat
-------------------------------------------------------------
$L$: suggest $(\forall \zeta \in \{a, f(a), f(f(a)), \ldots, x\}) \alpha(\zeta, y)$ .
```

Given a loop candidate $\alpha(x,y)$, we determine the set of values that the variable x takes on. Then we have as a new candidate for a loop invariant that α still holds for all the intermittent values between the initial value a and the current value x. For example, if $a \in Z$ and $f(x)=x+1$, then we get the candidate

$$L: \textbf{suggest} \ (\forall \zeta \in Z)(a \le \zeta \le x)\alpha(\zeta,y) \ .$$

This is a candidate and not an invariant since the program segment P may vary the value of y in such a way as to destroy the relation $\alpha(x,y)$ for previous values of x.

Note that a candidate invariant must sometimes be replaced by a stronger candidate in order to prove invariance. This is analogous to other forms of proof by induction, where it is often necessary to strengthen the desired theorem to carry out a proof. The reason is that by strengthening the theorem to be proved, we are at the same time strengthening the hypothesis that is used in the inductive step. We could not, for example, directly prove that the relation $(r \ge d) \vee (r=c-q \cdot d)$ is a loop invariant (that is the necessary condition for $r=c-q \cdot d$ to hold after the loop), since this candidate is not preserved by the loop, i.e.

$$[\ r \ge d \ \vee \ r=c-q \cdot d\ ] \wedge r \ge d \ \supset \ [\ r-d \ge d \ \vee \ r-d=c-(q+1) \cdot d\ ]$$

is not provable. On the other hand, we can prove that the stronger relation $r=c-q \cdot d$ is an invariant, since we have a stronger hypothesis on the left-hand side of the implication; that is,

$$r=c-q \cdot d \ \wedge \ r \ge d \ \supset \ r-d=c-(q+1) \cdot d$$

can be proved. Clearly, once we establish that $r=c-q \cdot d$ is an invariant, it follows that $(r \ge d) \vee (r=c-q \cdot d)$ also is.

Various specific methods of strengthening candidates have been discussed in the literature (Wegbreit [1974], Katz and Manna [1976], Moriconi [1974] and others); they are closely associated with methods of "top-down" structured programming. Related techniques are used by Greif and Waldinger [1974] and Suzuki and Ishihata [1977]. Also the candidates that Basu and Misra [1975] and Morris and Wegbreit [1977] derive, using the *subgoal-induction* method of verification, fall into this class.

6. *Counters*

A useful technique for proving certain properties of programs is the augmentation of a program with counters of various sorts. For example, by initializing a counter to zero upon entering a loop and incrementing it by one with each iteration, the value of the counter will indicate the number of times that the loop has been executed. Then, relations between the program variables and the counter can be found. By deriving upper/lower bounds on the counter, the termination of the loop may be proved and time complexity analyzed.

As a simple example, reconsider our (now annotated) division program

```
assert $c \in N$, $d \in N+1$, $q \in N$, $r=c-q \cdot d$ in
$P_0$: begin comment integer-division program
    $(q,r) := (0,c)$
    loop $L_0$: assert $q \leq c/d$
        until $r<d$
        $(q,r) := (q+1, r-d)$
        repeat
    $E_0$: assert $r<d$
    end .
```

The variable q is incremented by 1 with each loop iteration and is initialized to 0 ; thus, it serves as a loop counter. Since the loop invariant $q \leq c/d$ gives an upper bound on the value of the counter, and the counter is incremented with each loop iteration, the loop must terminate. Since the output invariant $r<d$ and global invariant $r=c-q \cdot d$ yield a lower bounds on the value of the counter, one can determine the total number of loop iterations.

Examples of the use of counters for proving termination have appeared in Katz and Manna [Dec. 1975] and Luckham and Suzuki [1977]. Loop counters may also be used to discover relations between variables by solving first-order difference equations. (See, for example, Elspas [1974] and Katz and Manna [1976]; Netzer [1976] applies this technique to recursive programs). Related work, making use of a small collection of "loop-plans" to decompose program loops, may be found in Waters [1977]. McCarthy and Talcott [1978] distinguish between *extensional* properties of programs that depend only on the function computed by the program and *intentional* properties, such as space and time requirements, that may be made explicit in derived programs containing counters.

In the following section, we demonstrate how two nontrivial programs can be annotated using the annotation rules. These examples are taken from the program annotation literature in order to demonstrate the power of our approach.

3. EXAMPLES

Our first example is the annotation of a program intended to divide two real numbers. The second example is a program with nested loops designed to sort an array.

Example 1: *Real Division.*

Consider the following program P_1 purporting to approximate the quotient c/d of two nonnegative real numbers c and d, where $c<d$. Upon termination, the variable q should be no greater than the exact quotient, and the difference between q and the quotient must be less than a given positive tolerance ϵ. The program, with its specifications included as assertions, is:

```
P1: begin comment real-division program
    B1: assert  0≤c<d,  0<ε
    (q, qq, r, rr) := (0, 0, 1, d)
    loop L1: assert  . . .
          until  r≤ε
          if  qq+rr≤c  then  (q, qq) := (q+r, qq+rr)  fi
          (r, rr) := (r/2, rr/2)
          repeat
    E1: suggest  q≤c/d,  c/d<q+ε
    end .
```

Our goal is to find loop invariants at L_1 in order to verify the output candidates at E_1. In our presentation of the annotation of this program, we first apply the assignment rules and then the control rules combined with a heuristic rule.

1. *Assignment Rules*

As a first step, we attempt to derive simple invariants by ignoring the control structure of the program, and considering only the assignment statements. This will yield global invariants that hold throughout execution.

We first look for range invariants by considering all assignments to each variable. For example, since the assignments to r are

$r := 1 \qquad r := r/2$,

we can apply the *multiplication rule*

$$\frac{x := a_0 \quad x := x{\cdot}a_1 \quad x := x{\cdot}a_2 \quad \ldots \quad \textbf{in } P}{\textbf{assert } x{\in}a_0{\cdot}a_1^{\mathsf{N}}{\cdot}a_2^{\mathsf{N}}{\cdot} \ldots \quad \textbf{in } P.}$$

Taking 1 for a_0 and 1/2 for a_1, we derive the global invariant

$$\textbf{assert } r{\in}1/2^{\mathsf{N}} \textbf{ in } P_1 . \qquad (1)$$

In other words, $r=1/2^n$ for some nonnegative integer n . From this it is possible to derive lower and upper bounds on r , i.e. $0<r\le1$, since $r=1$ when $n=0$, while $r=1/2^n$ approaches 0 as n grows large.

Similarly, applying the *multiplication rule* to the assignments to rr ,

$rr := d \qquad rr := rr/2$,

yields

$$\textbf{assert } rr{\in}d/2^{\mathsf{N}} \textbf{ in } P_1 . \qquad (2)$$

Since we are given that $d>0$, it follows that $0<rr\le d$.

The assignments to q are

$q := 0 \qquad q := q+r$.

Since we know (1) $r{\in}1/2^{\mathsf{N}}$, these assignments may be interpreted as the nondeterministic assignments

$q :\in 0 \qquad q :\in q+1/2^{\mathsf{N}}$

Using the *set-addition rule* <>

$$\frac{x :\in S_0 \quad x :\in x+S_1 \quad x :\in x+S_2 \quad \ldots \quad \textbf{in } P}{\textbf{assert } x{\in}S_0+\Sigma S_1+\Sigma S_2+ \ldots \quad \textbf{in } P \ ,}$$

we conclude

$$\textbf{assert } q{\in}\Sigma 1/2^{\mathsf{N}} \textbf{ in } P_1 .$$

This invariant states that q is a finite sum of elements of the form $1/2^n$, where n is some nonnegative integer. Since for any two such elements, one is a multiple of the other, it follows that the sum is of the form $m/2^n$, where $m, n \in \mathbf{N}$:

$$\textbf{assert}\ \ q \in \mathbf{N}/2^{\mathbf{N}}\ \ \textbf{in}\ P_1 \tag{3}$$

(i.e. q is a dyadic rational number).

From (2) $rr \in d/2^{\mathbf{N}}$ and the assignments

$$qq := 0 \qquad qq := qq + rr\ ,$$

we get by the same *set-addition rule*

$$\textbf{assert}\ \ qq \in d \cdot \Sigma 1/2^{\mathbf{N}}\ \ \textbf{in}\ P_1\ ,$$

or

$$\textbf{assert}\ \ qq \in d \cdot \mathbf{N}/2^{\mathbf{N}}\ \ \textbf{in}\ P_1\ . \tag{4}$$

The above four invariants give the range of each of the four program variables. Now we take up relations between pairs of variables by considering their respective assignments. Consider, first, the variables r and rr. Their assignments are

$$(r, rr) := (1, d) \qquad (r, rr) := (r/2, rr/2)\ .$$

Each time one is halved, so is the other; therefore, the proportion between the initial values of r and rr is maintained throughout loop execution. This is an instance of the *multiplication-relation rule*

$$\frac{\begin{array}{l}(x,y) := (a_0, b_0) \quad (x,y) := (x \cdot u^{a_1}, y \cdot u^{b_1}) \\ \qquad\qquad\qquad\qquad (x,y) := (x \cdot v^{a_1}, y \cdot v^{b_1}) \quad \ldots \quad \textbf{in}\ P\end{array}}{\textbf{assert}\ \ x^{b_1} \cdot b_0^{a_1} = a_0^{b_1} \cdot y^{a_1}\ \ \textbf{in}\ P\ ,}$$

yielding $r^1 \cdot d^1 = 1^1 \cdot rr^1$ which simplifies to

$$\textbf{assert}\ \ rr = d \cdot r\ \ \textbf{in}\ P_1\ . \tag{5}$$

(The rule may be matched with the assignments in the following manner: Clearly, the assignment $(x,y):=(a_0,b_0)$ matches with $(r,rr):=(1,d)$ by instantiating x, y, a_0, and b_0 with r, rr, 1, and d, respectively. Substituting these values in the second assignment of the rule, we are left with $(r,rr):=(r\cdot u^{a_1}, rr\cdot u^{b_1})$ to match with $(r,rr):=(r/2, rr/2)$. To match $r\cdot u^{a_1} \Rightarrow r/2$, divide both sides by r, leaving $u^{a_1} \Rightarrow 1/2$. This in turn is effected by instantiating $a_1 \Rightarrow 1$ and $u \Rightarrow 1/2$. It remains to match $rr\cdot(1/2)^{b_1} \Rightarrow rr/2$, i.e. $(1/2)^{b_1} \Rightarrow 1/2$, and b_1 instantiates to 1.)

The assignments to q and qq are

$$(q,qq) := (0,0) \qquad (q,qq) := (q+r, qq+rr) .$$

Using (5) $rr=d\cdot r$ to substitute for rr in the assignment $qq:=qq+rr$, we have

$$(q,qq) := (0,0) \qquad (q,qq) := (q+r, qq+d\cdot r) ,$$

which is an instance of the *addition-relation rule*

$$\frac{\begin{array}{ll}(x,y) := (a_0,b_0) & (x,y) := (x+a_1\cdot u, y+b_1\cdot u) \\ & (x,y) := (x+a_1\cdot v, y+b_1\cdot v) \quad \ldots \quad \textbf{in } P\end{array}}{\textbf{assert } a_1\cdot(y-b_0)=b_1\cdot(x-a_0) \textbf{ in } P .}$$

Thus we have the global invariant $1\cdot(qq-0)=d\cdot(q-0)$, i.e.

$$\textbf{assert } qq=d\cdot q \textbf{ in } P_1 . \tag{6}$$

In all, we have established the following global invariants:

$$\textbf{assert } r\in 1/2^{\mathsf{N}},\ rr\in d/2^{\mathsf{N}},\ q\in \mathsf{N}/2^{\mathsf{N}},\ qq\in d\cdot\mathsf{N}/2^{\mathsf{N}},\ rr=d\cdot r,\ qq=d\cdot q \textbf{ in } P_1 .$$

2. *Control Rules*

So far we have derived global invariants from the assignment statements, ignoring the control structure of the program. We turn now to local invariants extracted from the program structure.

By applying the *assignment axiom*

```
x := a
assert  x=a
```

to the multiple assignment at the beginning of the program we get the local invariant

```
assert  q=0,  qq=0,  r=1,  rr=d
```

just prior to the loop. The *loop axiom*,

```
loop P'
     until  t
     assert  ¬t
     P''
     repeat
assert  t
```

yields $r>e$ at the head of the loop body and $r\leq e$ at E_1. Thus far, we have the annotated program segment

```
assert  q=0,  qq=0,  r=1,  rr=d
loop L₁: assert  . . .
     until  r≤e
     assert  r>e
     if  qq+rr≤c  then  (q,qq) := (q+r,qq+rr)  fi
     (r,rr) := (r/2,rr/2)
     repeat
E₁: assert  r≤e .
```

Applying the *forward test rule*,

```
assert  α
if  t  then   L':
                P'
     else     L'':
                P''
     fi
─────────────────────
L': assert  α,  t
L'': assert  α,  ¬t ,
```

to the conditional statement of the loop,

```
if  qq+rr≤c  then  (q,qq) := (q+r,qq+rr)  fi
```

yields

```
if  $qq+rr \le c$  then assert  $r>e$,  $qq+rr \le c$
                          $(q,qq)$ := $(q+r, qq+rr)$
                    else  assert  $r>e$,  $c<qq+rr$
                    fi .
```

Using a variant of the *forward assignment rule*,

$$\frac{\begin{array}{l}\textbf{assert}\ \alpha(\bar{u},\bar{y})\\ \bar{x} := \bar{u}\\ L:\end{array}}{L\text{: }\textbf{assert}\ \alpha(\bar{x},\bar{y})}\ ,$$

where $\bar{x}$ does not appear in $\alpha(\bar{\xi},\bar{y})$, the assignment of the **then**-branch transform the invariant $qq+rr \le c$ into $qq \le c$ and leaves $r>e$ unchanged. We obtain

```
if  $qq+rr \le c$  then $(q,qq)$ := $(q+r, qq+rr)$
                         assert  $r>e$,  $qq \le c$
                    else  assert  $r>e$,  $c<qq+rr$
                    fi .
```

We may now apply the *forward branch rule*

$$\frac{\begin{array}{lll}\textbf{if}\ t & \textbf{then} & P'\\ & & \textbf{assert}\ \alpha\\ & \textbf{else} & P''\\ & & \textbf{assert}\ \beta\\ & \textbf{fi} & \\ L: & & \end{array}}{L\text{: }\textbf{assert}\ \alpha \vee \beta}\ ,$$

to the two possible outcomes of the conditional. We obtain the invariant

assert $(r>e \wedge qq \le c) \vee (r>e \wedge c<qq+rr)$,

which simplifies to just

assert $r>e$,

since $r>e$ appears in both disjuncts while $qq \le c \vee c<qq+rr$ is implied by the global invariant (2') $rr>0$ ($qq \le c \vee c<qq$ is a tautology, and if rr is positive, then $c<qq$ implies $c<qq+rr$).

By application of the *forward assignment rule*

$$\frac{\begin{array}{l}\textbf{assert}\ \alpha(x,y)\\ x := f(x,y)\\ L:\end{array}}{L:\ \textbf{assert}\ \alpha(f^{-}(x,y),y)}$$

to the invariant $r > e$, we get

```
assert  $2 \cdot r > e$
```

at the end of the loop. By applying the *forward loop-body rule*,

$$\frac{\begin{array}{l}\textbf{assert}\ \alpha\\ \textbf{loop}\ L:\\ \quad P\\ \quad \textbf{assert}\ \beta\\ \quad \textbf{repeat}\end{array}}{L:\ \textbf{assert}\ \alpha \vee \beta}\ ,$$

taking $2 \cdot r > e$ for β, we derive the loop invariant

$$L_1:\ \textbf{assert}\ (q{=}qq{=}0 \wedge r{=}1 \wedge rr{=}d) \vee 2 \cdot r > e\ .$$

In order to simplify the presentation, we shall use instead the weaker

$$L_1:\ \textbf{assert}\ r{=}1 \vee 2 \cdot r > e\ . \qquad (7)$$

3. *Heuristic Rules*

Recall that the control rules gave us

```
if  $qq+rr \le c$  then  $(q,qq) := (q+r, qq+rr)$
                       assert  $r>e,\ qq \le c$
                  else  assert  $r>e,\ c<qq+rr$
                  fi ,
```

but that the disjunction of $qq \le c$ and $c < qq+rr$ turned out to be a tautology. The *conditional heuristic*

```
if t then  P'
           assert α
     else  P''
           assert β
     fi
L:
─────────────────
L: suggest α, β
```

suggests that each of the two invariants, $qq \le c$ and $c < qq+rr$, that hold at the end of one of the conditional paths, may be an invariant for both paths. So we have the candidate

suggest $qq \le c,\ c < qq+rr$

following the conditional and preceding the assignment

$(r, rr) := (r/2, rr/2)$.

By application of the *forward assignment rule*

```
suggest γ(x,y)
x := f(x,y)
L:
─────────────────────
L: suggest γ(f⁻(x,y),y)
```

to the two candidates, we get

suggest $qq \le c,\ c < qq+2 \cdot rr$

at the end of the loop.

Finally, by applying the *forward loop-body rule*,

```
assert α
loop L:
     P
     suggest γ
     repeat
──────────────────
L: suggest α∨γ ,
```

we get the candidates

L_1: **suggest** $(q=qq=0 \wedge r=1 \wedge rr=d) \vee qq \le c,\ (q=qq=0 \wedge r=1 \wedge rr=d) \vee c < qq+2 \cdot rr$.

Both candidates may be simplified, since their first disjunct implies their second, leaving

L_1: **suggest** $qq \le c,\ c < qq + 2 \cdot rr$.

These two candidates can indeed be proved to be invariants: The first candidate, $qq \le c$, derived from the initialization and **then**-paths, is unaffected by the else-path which leaves the value of qq unchanged. Similarly, the other candidate, $c < qq + 2 \cdot rr$, derived from the initialization and **else**-paths, is maintained true by the **then**-path. So we have the loop invariants

L_1: **assert** $qq \le c,\ c < qq + 2 \cdot rr$. (8)

Note that we have not yet made any use of the candidates

E_1: **suggest** $q \le c/d,\ c/d < q + e$,

suggested by the output specification. For completeness, we shall apply a heuristic to these candidates, though no new invariants will be derived. The *top-down heuristic rule*

```
assert $\alpha$
loop $L$:
     until $t$
     $P$
     repeat
suggest $\gamma$

fact $\gamma$ when $\alpha$
─────────────────
$L$: suggest $\gamma$
```

suggests that the output candidate $q \le c/d$ may itself be a loop invariant, since it is true upon entering the loop. Indeed it is an invariant (it is implied by the loop invariant $qq \le c$ and the global invariant $qq = q \cdot d$). On the other hand, the second output candidate, $c/d < q + e$, does not even hold for the initialization path, when $q = 0$.

Since there are no assignments between the loop and the end of the program, all the loop invariants may be pushed forward unchanged, and hold upon termination. The output invariants include

E_1: **assert** $(r = 1 \vee 2 \cdot r > e),\ qq \le c,\ c < qq + 2 \cdot rr,\ r \le e$. (9)

These invariants, along with the global invariants

assert $rr = d \cdot r,\ qq = d \cdot q$ **in** P_1 ,

imply $q \le c/d$ as specified. However, they do *not* imply $c/d < q + e$, only $c/d < q + 2 \cdot e$. In fact, our program as given is incorrect. In another chapter, we have already seen how such invariants may be used to guide the debugging of the program.

4. *Loop Counters*

By introducing an imaginary loop counter n — initialized to 0 upon entering the loop and incremented by 1 with each iteration — one may derive relation between the program variables and the number of iterations.

The extended program, annotated with some of the invariants we have already found, is:

```
assert $rr=d\cdot r,\ qq=d\cdot q,\ r\in 1/2^{N},\ rr\in N/2^{N}$ in
$P_1$: begin comment extended real-division program
    $B_1$: assert $0\le c<d,\ 0<e$
    $(q,qq,r,rr)$ := $(0,0,1,d)$
    $n$ := $0$
    loop $L_1$: assert $(r=1\vee 2\cdot r>e),\ qq\le c,\ c<qq+2\cdot rr$
        until $r\le e$
        if $qq+rr\le c$ then $(q,qq)$ := $(q+r,qq+rr)$ fi
        $(r,rr)$ := $(r/2,rr/2)$
        $n$ := $n+1$
        repeat
    $E_1$: assert $(r=1\vee 2\cdot r>e),\ qq\le c,\ c<qq+2\cdot rr,\ r\le e$
    end .
```

Obviously, we may

assert $n\in N$ **in** P_1 . (10)

For the variables r and n, we have the assignments

$$(r,n) := (1,0) \qquad (r,n) := (r/2,n+1)$$

and we can apply the *linear rule*

$$\frac{(x,y) := (a_0,b_0) \quad (x,y) := (a_1\cdot x+a_2,\ y+b_2) \ \textbf{in}\ P}{\textbf{assert}\ [x\cdot(a_1-1)+a_2]^{b_2}\cdot a_1^{b_0}=[a_0\cdot(a_1-1)+a_2]^{b_2}\cdot a_1^{y} \ \textbf{in}\ P} .$$

With this rule we get the global invariant

$$\textbf{assert}\ [r\cdot(1/2-1)+0]^{1}\cdot(1/2)^{0}=[1\cdot(1/2-1)+0]^{1}\cdot(1/2)^{n} \ \textbf{in}\ P_1 .$$

which simplifies to

assert $r=1/2^n$ **in** P_1 (11)

Applying the same rule to the assignments

$(rr, n) := (d, 0) \qquad (rr, n) := (rr/2, n+1)$

we deduce

assert $rr=d/2^n$ **in** P_1 . (12)

With these loop-counter invariants, the total number of loop iterations as a function of the input values may be determined. Using (11), we can substitute $1/2^n$ for r in the loop invariant (7) $r=1 \vee 2 \cdot r > e$ and in the output invariant (9) $r \le e$, and get $1/2^n=1 \vee 2/2^n > e$ at L_1 and $1/2^n \le e$ at E_1 . Taking the logarithm (e is positive), we have the upper bound

$n=0 \vee n < -log_2 e+1$

and lower bound

$-log_2 e \le n$

on the number of loop iterations n . Note that by finding a loop invariant giving an upper bound on the number of iterations, we have actually proved that the loop terminates.

Combining both bounds at E_1 gives (assuming $n \ne 0$)

$-log_2 e \le n < -log_2 e+1$,

or, since n is an integer (10), it is equal to the one integer lying between its lower and upper bound $-\lfloor log_2 e \rfloor$. Thus we have the output invariant

E_1: **assert** $n=0 \vee n=-\lfloor log_2 e \rfloor$. (13)

Since n is the number of times the loop was executed before termination, we have derived the desired expression for the time complexity of the loop.

Example 2: *Selection Sort.*

The previous example contained only one loop and dealt with simple variables. As a more challenging example, we annotate an array-manipulation program containing nested loops. The program is intended to sort the array $A[0:n]$ of $n+1$ elements $A[0]$, $A[1]$, . . . , $A[n]$ in ascending sequence. The output specification can therefore be expressed as

$$(\forall \zeta)(0 \le \zeta < n)(A[\zeta] \le A[\zeta+1]) \ \wedge \ bag(A[0:n]) = bag(A_{B_2}[0:n])$$

where $bag(A[0:n]) = bag(A_{B_2}[0:n])$ means that the multiset (bag) of elements in the array segment $A[0:n]$ is equal to the multiset of elements in the initial value of the array A_{B_2} ,. i.e. $A[0:n]$ is a permutation of $A_{B_2}[0:n]$. The program is:

```
P₂: begin comment selection-sort program
    B₂: assert n∈N
    i := 0
    loop L₂: assert  . . .
          until i≥n
          P₃: begin
                (j, m, k) := (i+1, A[i], i)
                loop  L₃: assert  . . .
                      until j>n
                      if A[j]<m then (m, k) := (A[j], j) fi
                      j := j+1
                      repeat
                (A[k], A[i], i) := (A[i], m, i+1)
                end
          repeat
    E₂: suggest (∀ζ)(0≤ζ<n)(A[ζ]≤A[ζ+1]), bag(A[0:n])=bag(A_B₂[0:n]).
    end
```

1. *Assignment Rules*

We first try to determine the range of the program variables. The variables in the program P_2 are i , j , k , m , and A ; the inner loop (the program segment P_3) sets the variables j , k and m , and leaves i and A unchanged.

The assignments to i are

$i := 0 \qquad i := i+1$,

which by the *addition rule* gives the global invariant

assert $i \in \mathrm{N}$ **in** P_2 . (1)

The assignments to j are

$$j := i+1 \qquad j := j+1 \ .$$

Since we know $i \in \mathbb{N}$, we may substitute $\mathbb{N}$ for i to obtain the nondeterministic assignments

$$j :\in \mathbb{N}+1 \qquad j :\in j+1 \ ,$$

and by the *set-addition rule* we get $j \in \mathbb{N}+1+\Sigma 1$, which simplifies to

$$\textbf{assert } j \in \mathbb{N},\ 1 \leq j \textbf{ in } P_2 \ . \qquad (2)$$

(Recall that these global invariants only hold after $j:=i+1$ is executed for the first time.) Since within P_3 the value of i is unchanged, it may be regarded as constant. We can therefore apply the *addition rule* to the assignments to j, $j:=i+1$ and $j:=j+1$, obtaining

$$\textbf{assert } j \in i+1+\mathbb{N} \textbf{ in } P_3$$

and consequently

$$\textbf{assert } i<j \textbf{ in } P_3 \ . \qquad (3)$$

The assignments to k are

$$k := i \qquad k := j \ .$$

Using (1) and (2) to substitute $\mathbb{N}$ for i and j, we have

$$k :\in \mathbb{N} \qquad k :\in \mathbb{N}$$

and from the simple *set-union rule*

$$\frac{x :\in S_0 \quad x :\in S_1 \textbf{ in } P}{\textbf{assert } x \in S_0 \cup S_1 \textbf{ in } P}$$

it follows that

$$\textbf{assert } k \in \mathbb{N} \textbf{ in } P_2 \ . \qquad (4)$$

In P_3, as we have seen, i is constant and $j \in i+1+\mathbb{N}$, so we substitute $i+1+\mathbb{N}$ for j in the assignments to k to obtain

$$k :\in i \qquad k :\in i+1+\mathbb{N}$$

By the same *set-union rule*, we have that k belongs to the union of i and $i+1+\mathbb{N}$. Therefore $k \in i+\mathbb{N}$, and

assert $i \leq k$ **in** P_3 . (5)

Finally, for m we have the assignments

$m := A[i] \qquad m := A[j]$.

Using (1) $i \in \mathbb{N}$ and (2) $j \in \mathbb{N}$ to substitute $\mathbb{N}$ for i and j, we get

$m :\in A[\mathbb{N}] \qquad m :\in A[\mathbb{N}]$.

Thus, by the *set-union rule*, we obtain

assert $m \in A[\mathbb{N}]$ **in** P_2 . (6)

In the following subsections, we shall apply the control rules and heuristics first to the inner loop and then to the outer loop.

2. *Control Rules - Inner Loop*

The inner loop of the program is

```
(j, m, k) := (i+1, A[i], i)
loop L3: assert . . .
     until j>n
     if A[j]<m then (m,k) := (A[j],j) fi
     j := j+1
     repeat .
```

At any point in a program, the disjunction of what is known from the paths leading to that point is an invariant. We shall use the control rules to obtain loop invariants at label L_3, by considering the three paths leading to L_3: the initialization path from L_2 to L_3, the loop-body path from L_3 to L_3 via the **then**-branch of the conditional, and the loop-body path via the **else**-branch of the conditional.

From the initialization path, we have upon entering the inner loop

$i<n \wedge j=i+1 \wedge m=A[i] \wedge k=i$. (7)

The conjunct $i<n$ derives from the negation of the outer-loop exit test, using the *loop axiom*

```
loop $P'$
     until $t$
     assert $\neg t$
     $P''$
     repeat
assert $t$
```

By applying the *assignment axiom*

```
x := a
assert $x=a$
```

to the assignment of the initialization path

$$(j,m,k) := (i+1, A[i], i) ,$$

we obtain the three invariants $j=i+1$, $m=A[i]$ and $k=i$.

At the head of the inner-loop body, we have the invariant

$$j \le n \ \wedge\ i=i_{L_5} \ \wedge\ A=A_{L_5} \ \wedge\ j=j_{L_5} \ \wedge\ k=k_{L_5} \ \wedge\ m=m_{L_5} ,$$

where x_L , for some variable x and label L , denotes the value of x when control was last at L . The first conjunct is the negation of the exit test and the other conjuncts, which are generated at L_5 using the *label axiom*,

```
$L$: assert $x=x_L$ ,
```

have been pushed passed the exit test unchanged. This is an application of the *forward loop-exit rule*

```
loop $P'$
     assert $\alpha$
     until $t$
     $L'$:
     $P''$
     repeat
$L''$:
____________________
$L'$: assert $\alpha$, $\neg t$
$L''$: assert $\alpha$, $t$
```

to the inner loop. After executing the assignment in the **then**-branch of the conditional, we know

$$j \le n \ \wedge\ m = A[j] \ \wedge\ k = j \ \wedge\ i = i_{L_3} \ \wedge\ A = A_{L_3} \ \wedge\ j = j_{L_3} \ .$$

The second and third conjuncts derive from the assignments (by the *assignment axiom*); all the other conjuncts have been propagated forward by the *forward test rule*

```
assert  $\alpha$
if  $t$  then    $L'$:
                $P'$
        else    $L''$:
                $P''$
        fi
______________________
$L'$: assert  $\alpha$, $t$
$L''$: assert  $\alpha$, $\neg t$
```

and *forward assignment rule*

```
assert  $\alpha(x,y)$
$x$ := $f(x,y)$
$L$:
______________________
$L$: assert  $\alpha(f^{-}(x,y),y)$ .
```

After the (empty) else-branch of the conditional, we have

$$j \le n \ \wedge\ m \le A[j] \ \wedge\ i = i_{L_3} \ \wedge\ A = A_{L_3} \ \wedge\ j = j_{L_3} \ \wedge\ k = k_{L_3} \ \wedge\ m = m_{L_3} \ .$$

The second conjunct is the negation of the conditional test; it is derived from the *conditional axiom*

```
if  $t$  then    assert  $t$
                $P'$
        else    assert  $\neg t$
                $P''$
        fi .
```

Since we must have traversed either the then- or else-branch, we know by the *forward branch rule*

```
if  $t$  then    $P'$
                assert  $\alpha$
        else    $P''$
                assert  $\beta$
        fi
$L$:
______________________
$L$: assert  $\alpha \vee \beta$
```

that after the conditional

$$\begin{aligned}&(\ j\le n \ \wedge\ m=A[j] \ \wedge\ k=j \ \wedge\ i=i_{L_3} \ \wedge\ A=A_{L_3} \ \wedge\ j=j_{L_3}\ )\\ \vee\ &(\ j\le n \ \wedge\ m\le A[j] \ \wedge\ i=i_{L_3} \ \wedge\ A=A_{L_3} \ \wedge\ j=j_{L_3} \ \wedge\ k=k_{L_3} \ \wedge\ m=m_{L_3}\ )\ .\end{aligned}$$

Thus, at the end of the loop body, after incrementing j by 1, we have (by the *forward assignment rule*)

$$\begin{aligned}&(\ j-1\le n \ \wedge\ m=A[j-1] \ \wedge\ k=j-1 \ \wedge\ i=i_{L_3} \ \wedge\ A=A_{L_3} \ \wedge\ j-1=j_{L_3}\ )\\ \vee\ &(\ j-1\le n \ \wedge\ m\le A[j-1] \ \wedge\ i=i_{L_3} \ \wedge\ A=A_{L_3} \ \wedge\ j-1=j_{L_3} \ \wedge\ k=k_{L_3} \ \wedge\ m=m_{L_3}\ )\ .\end{aligned} \tag{8}$$

Furthermore, if a relation α holds upon entering a loop, and we know that the loop body either does not change the values of the variables in α, or reachieves α for the new values of the variables, then α is a loop invariant. This is the *protected-invariant rule*

```
assert  $\alpha(x)$
loop $L$:
     $P$
     assert  $\alpha(x) \vee x=x_L$
     repeat
_______________________
$L$: assert  $\alpha(x)$ .
```

By substituting k for $j-1$ in the first disjunct of (8), we may derive $k\le n$ and $m=A[k]$. Thus, at the end of the loop body we know $(k\le n \wedge m=A[k]) \vee (A=A_{L_3} \wedge k=k_{L_3} \wedge m=m_{L_3})$. This invariant is of the form $\alpha(x) \vee x=x_L$, taking $\alpha(x)$ to be $k\le n \wedge m=A[k]$ and x to be the variables A, k and m. The first disjunct indicates that the **then**-path achieves $\alpha(x)$; the second disjunct states that the **else**-path leaves A, k and m unchanged. From invariant (7) preceding the loop, we can derive that initially $k\le n$ and $m=A[k]$. So we have

L_3: **assert** $k\le n,\ m=A[k]$. (9)

Similarly, by (8) we have $i=i_{L_3}$ for both loop-body paths, and by (7) we have $i<n$ upon entering the loop. Taking $\alpha(i)$ to be $i<n$, we get

L_3: **assert** $i<n$. (10)

Disjoining invariant (7) of the initialization path and (8) from the loop-body path, we get the following inner-loop invariant (by the *forward loop-body rule*):

$$\begin{aligned}L_3\text{: \textbf{assert} } &(\ i<n \ \wedge\ j=i+1 \ \wedge\ m=A[i] \ \wedge\ k=i\ )\\ \vee\ &(\ j-1\le n \ \wedge\ m=A[j-1] \ \wedge\ k=j-1\ )\\ \vee\ &(\ j-1\le n \ \wedge\ m\le A[j-1]\ )\end{aligned} \tag{11}$$

(The conjuncts refering to the previous value of a variable at L_5 have been removed.)

Now we extract the "common denominator" of the disjuncts in (11) arising from the different paths. The relation $j-1 \le n$ appears in the second two disjuncts and is implied by the two conjuncts $i<n$ and $j=i+1$ of the first disjunct, so we get the invariant

$$L_5\text{: assert } j-1 \le n \ . \tag{12}$$

In the first disjunct of (11) we have $j=i+1 \wedge m=A[i]$, in the second we have $m=A[j-1]$, while in the third we have $m \le A[j-1]$, thus for all paths

$$L_5\text{: assert } m \le A[j-1] \ . \tag{13}$$

Alternatively, we could have used the *conditional heuristic* rule, rather than the *protected-invariant rule*, to generate these assertions. The heuristic, however, would have yielded candidates requiring further verification.

3. *Generalization Heuristic - Inner Loop*

The *generalization heuristic* rule is particularly valuable for loops involving arrays:

```
assert $x=a$
loop $L$: assert $\alpha(x,y)$
     $P$
     assert $x=x_L+1$
     repeat
─────────────────────────────
$L$: suggest $(\forall\zeta)(a\leq\zeta\leq x)\alpha(\zeta,y)$ .
```

To apply this heuristic, reconsider the inner-loop invariant (13) $\alpha(j,m)$: $m\leq A[j-1]$ at L_3. Initially j is $i+1$, and at the end of the loop body $j=j_{L_3}+1$, so, as an invariant candidate, we try

L_3: **suggest** $(\forall\zeta)(i+1\leq\zeta\leq j)(m\leq A[\zeta-1])$,

which we shall abbreviate as $m\leq A[i:j-1]$. Checking the candidate for the **then**- and **else**-paths determines that it is in fact an invariant; thus, we have for the inner loop

L_3: **assert** $m\leq A[i:j-1]$. (14)

So far we have derived the following inner-loop invariants

L_3: **assert** $k\leq n$, $m=A[k]$, $i<n$, $j-1\leq n$, $m\leq A[i:j-1]$.

We turn now to consider the outer loop.

4. *Control Rules - Outer Loop*

Using the *forward loop-exit rule* , the invariants at L_3 may be propagated past the exit test $j>n$, obtaining

assert $k\leq n$, $m=A[k]$, $i<n$, $j-1\leq n$, $m\leq A[i:j-1]$, $j>n$

just prior to the assignment

$(A[k],A[i],i)$:= $(A[i],m,i+1)$.

Propagating these invariants past the assignment, we get the following invariants at the end of the outer-loop body:

assert $k\leq n$, $i\leq n$, $m\leq A[i:j-1]$, $m=A[i-1]$, $j-1=n$. (15)

The invariant $k \le n$ is propagated unchanged. The invariant $i < n$ becomes $i-1 < n$ after executing $i := i+1$ (by the *forward assignment rule*), which is equivalent to $i \le n$ (since both i and n are integers). The invariant $m \le A[i:j-1]$ still holds after assigning $A[i]$ to $A[k]$, since $m \le A[i]$ held before and consequently $m \le A[k]$ holds now as well; however, after the assignment to $A[i]$, it becomes $m \le A[i+1:j-1]$. (To propagate invariants over an array assignment, there is a *forward array-assignment rule*

$$\frac{\begin{array}{l}\textbf{assert}\ \alpha(A,z)\\ A[y]\ :=\ f(A[y],z)\\ \underline{L:}\end{array}}{L\text{: }\textbf{assert}\ \alpha(assign(A,y,f^{-}(A[y],z)),z)}\ ,$$

where the array function $assign(A,y,z)$ yields A with z replacing $A[y]$, and $f^{-}(f(A[y],z),z) = A[y]$. This rule states that after the assignment the invariant still holds for all the elements of A , save $A[y]$; it also holds for old value of $A[y]$, $f^{-}(A[y],z)$. In our case, the old value of $A[i]$ cannot be reconstructed, so the index i is removed from the range $[i:j-1]$.) After incrementing i , $m \le A[i+1:j-1]$ becomes $m \le A[i:j-1]$. The assignment $A[i] := m$ generates the invariant $m = A[i]$ (by the *assignment axiom*), which becomes $m = A[i-1]$ after incrementing i . Finally, the invariants $j-1 \le n$ and $j > n$ simplify to $j-1 = n$ (since (2) $j \in \mathbb{N}$).

Clearly upon entering the outer loop (by the *assignment axiom*)

$$i=0\ .$$

Thus, by the *forward loop-body rule* , we have the outer-loop invariant

$$L_2\text{: }\textbf{assert}\ i=0\ \vee\ (k \le n \wedge i \le n \wedge m \le A[i:j-1] \wedge m = A[i-1] \wedge j-1=n)$$

with the following two corollaries:

$$L_2\text{: }\textbf{assert}\ i=0 \vee A[i-1] \le A[i:n] \qquad (16)$$

(the second disjunct follows from $m \le A[i:j-1]$, $m = A[i-1]$ and $j-1=n$), and

$$L_2\text{: }\textbf{assert}\ i \le n \qquad (17)$$

(since $i=0$ implies $i \le n$ for $n \in \mathbb{N}$). If we use the *forward loop-exit rule* to push $i \le n$ past the exit test $i \ge n$ and out of the loop, we get the output invariant $i \le n \wedge i \ge n$ at E_2 , or,

$$E_2\text{: }\textbf{assert}\ i=n\ . \qquad (18)$$

5. *Heuristics - Outer Loop*

We use the *generalization heuristic rule* to generalize (16) for the counter i, where $\alpha(i, A)$ is $i=0 \vee A[i-1] \le A[i:n]$. Since i is initially 0, this yields the candidate

L_2: **suggest** $(\forall \zeta)(0 \le \zeta \le i)(\zeta=0 \vee A[\zeta-1] \le A[\zeta:n])$.

This is equivalent to

L_2: **suggest** $(\forall \zeta)(0 \le \zeta < i)(A[\zeta] \le A[\zeta+1:n])$

and states, in effect, that the array elements $A[0:i-1]$ are sorted and that they are all smaller than the array elements $A[i:n]$. Though the array A is modified along the inner-loop exit path by the assignments

$(A[k], A[i]) := (A[i], m)$,

using $i \le k \le n$ and $m=A[k]$ (from (5) and (9)), invariance along that path can be shown.

Since $i \le k$, the assignment to $A[k]$ cannot destroy the order of $A[0:i-1]$, and clearly an assignment to $A[i]$ has no effect. Since both i and k are in the range $[i:n]$, the candidate implies that $A[0:i-1] \le A[i]$ and $A[0:i-1] \le A[k]$. So assigning $A[i]$ to $A[k]$ does not affect the relation. Lastly, since m is equal to the previous value of $A[k]$, assigning that value to $A[i]$ also preserves the invariant. The effect is to exchange the values of $A[k]$ and $A[i]$.

So we have the outer-loop invariant

L_2: **assert** $(\forall \zeta)(0 \le \zeta < i)(A[\zeta] \le A[\zeta+1:n])$. (19)

This may be pushed out of the loop to E_2, and with (18), i.e. $i=n$ at E_2, implies the first conjunct of the output specification,

$(\forall \zeta)(0 \le \zeta < n)(A[\zeta] \le A[\zeta+1])$.

The *top-down heuristic rule*

```
assert $\alpha$
loop $L$:
    until $t$
    $P$
    repeat
suggest $\gamma$
```

$$\frac{\textbf{fact}\ \gamma\ \textbf{when}\ \alpha}{L\text{: }\textbf{suggest}\ \gamma}$$

suggests that the output specification $bag(A[0:n])=bag(A_{B_2}[0:n])$, which is obviously true initially, is itself a candidate at L_2. Since the assignments to A have the effect of exchanging the values of $A[k]$ and $A[i]$, we have the invariant

$$L_2\text{: }\textbf{assert}\ bag(A[0:n])=bag(A_{B_2}[0:n])\ . \qquad (20)$$

6. *Loop Counters*

To determine the time complexity of this program, we add three counters: one for the outer loop (the variable i is effectively an outer-loop counter), one for the inner loop (call it n_3), and a third to sum the total number of inner-loop executions (call it n^*).

The extended program, annotated with some of the more important loop and output assertions, is:

```
assert $i \in \mathbb{N}$ in
$P_2$: begin comment extended selection-sort program
    $B_2$: assert $n \in \mathbb{N}$
    $i$ := 0
    $n^*$ := 0
    loop $L_2$: assert $i \le n$, $(\forall \zeta)(0 \le \zeta < i)(A[\zeta] \le A[\zeta:n])$, $bag(A[0:n]) = bag(A_{B_2}[0:n])$
        until $i \ge n$
        assert $j, k \in \mathbb{N}$ in
        $P_3$: begin
            $(j, m, k)$ := $(i+1, A[i], i)$
            $n_3$ := 0
            loop $L_3$: assert $i < n$, $i < j \le n+1$, $i \le k \le n$, $m = A[k]$, $m \le A[i:j-1]$
                until $j > n$
                if $A[j] < m$ then $(m, k)$ := $(A[j], j)$ fi
                $j$ := $j+1$
                $n_3$ := $n_3+1$
                repeat
            $(A[k], A[i], i)$ := $(A[i], m, i+1)$
            $n^*$ := $n^*+n_3$
            end
        repeat
    $E_2$: assert $i = n$, $(\forall \zeta)(0 \le \zeta < i)(A[\zeta] \le A[\zeta+1:n])$, $bag(A[0:n]) = bag(A_{B_2}[0:n])$
    end .
```

By the *addition-relation rule*, we can easily determine that n_3 is equal to $j-i-1$, since j is initialized to $i+1$ and is incremented by 1. We know from (15) that $j=n+1$ when the inner loop is left, and it follows that the inner loop is executed $n-i$ times for each outer-loop iteration. With each outer-loop iteration, i.e. each time i is incremented by 1, the total number of inner-loop iterations increases by $n-i$. Using the following *quadratic rule*

$$\frac{(x,y) := (a_0, b_0) \quad (x,y) := (x+a_2, y+b_1 \cdot x+b_2) \ \textbf{in}\ P}{\textbf{assert}\ (y-b_0)\cdot 2\cdot a_2{}^2 = (x-a_0)\cdot[b_1\cdot(x+a_0-a_2)+2\cdot a_2\cdot(b_2+b_1\cdot a_0)]\ \textbf{in}\ P}\ ,$$

and taking x to be i (initially 0 and incremented by 1) and y to be the total number of inner-loop executions (incremented by $n-i$), it may be determined that

$y \cdot 2 = i \cdot [-1 \cdot (i-1) + 2 \cdot n]$ is an invariant. (Recall the use in the previous chapter, page 122, of the inverse of this rule.) We have already seen that upon termination $i=n$, i.e. the outer loop is iterated a total of n times. Therefore, when the outer loop is left, $n^* = n \cdot (n+1)/2$, i.e. the total number of inner-loop executions is $n \cdot (n+1)/2$.

In a sense, annotating programs is "putting the cart before the horse" as the whole tenor of "structured programming" stresses developing invariants hand in hand with the code, and not ex post facto, as annotation implies. Nevertheless, the development of automatic annotation systems is important for a number of reasons:

● The real world contains many undocumented, underdocumented, and misdocumented programs. Even annotated programs appearing in structured-programming textbooks have fallen prey to error. A system that could help in documenting such programs would clearly be of utility.

● Ultimately, it is the responsibility of the programmer to guarantee the correctness of his product. Even if he uses one of the current automatic verification systems, he is required to supply most, if not all, of the necessary invariant assertions. The goal of automatic program annotation is to relieve the programmer of this burden. Agreed, no present or foreseeable system is likely discover very subtle invariants, or those based on deep mathematical theorems, but such invariants are likely to be uppermost in the programmer's mind anyway. It is the "obvious" invariants that he finds annoying to have to formulate, and indeed often forgets, causing the system to fail in its proof. For example, the invariant $k \leq n$ is crucial for the correctness of the selection sort program; if the programmer omits it, the verification system will not be able to prove correctness. Fortunately, it is just these invariants that an automatic annotation system would find easy to derive. Similarly, invariants needed to demonstrate the absence of runtime errors are usually quite simple, and there has already been some success in providing current verification systems with the capability of generating them.

● Annotation research attempts to formalize the intuitions that lie behind well-designed programs; thus, it has important implications for automatic program synthesis. In fact, the same rules that we used to generate invariants from programs may be inverted to generate programs from invariants.

● Annotation techniques may be used to discover important properties of programs other than correctness. For example, one may wish to analyze the complexity of an algorithm or compare the efficiency of two correct programs. This is not usually the programmer's responsibility. Indeed, even simple programs are sometimes very difficult to analyze (cf. Jonassen and Knuth [1978]).

REFERENCES

Balzer, R.M., N. Goldman, and D. Wile [Aug. 1977], *Informality in program specifications*, Proc. 5th Intl. Joint Conf. on Artificial Intelligence, Cambridge, MA, pp. 389-397.

Basu S. and J. Misra [March 1975], *Proving loop programs*, IEEE Software Engineering, vol. SE-1, no. 1, pp. 76-86.

Biermann, A.W. [1976], *Approaches to automatic programming*, Advances in Computers, vol. 15, Academic Press, New York, NY, pp. 1-63.

Boyer, R.S., B. Elspas, and K.N. Levitt [Apr. 1975], *SELECT—a formal system for testing and debugging programs by symbolic execution*, Proc. Intl. Conf. on Reliable Software, Los Angeles, CA, pp. 234-245.

Boyer, R.S. and J S. Moore [Jan. 1975], *Proving theorems about LISP functions*, JACM, vol. 22, no. 1, pp. 129-144.

Brown, R. [Oct. 1976], *Reasoning by analogy*, Working paper 132, Artificial Intelligence Laboratory, MIT, Cambridge, MA.

Buchanan, J.R. and D.C. Luckham [Mar. 1974], *On automating the construction of programs*, Memo AIM-236, Artificial Intelligence Laboratory, Stanford Univ., Stanford, CA.

Burstall, R.M. and J. Darlington [Jan. 1977], *A transformation system for developing recursive programs*, JACM, vol. 24, no. 1, pp. 44-67.

Caplain, M. [Apr. 1975], *Finding invariant assertions for proving programs*, Proc. Intl. Conf. on Reliable Software, Los Angeles, CA, pp. 165-171.

Chen, T.W. and N.V. Findler [Dec. 1976], *Toward analogical reasoning in problem solving by computers*, Technical report 115, Dept. of Computer Science, State Univ. of New York, Buffalo, NY.

Conway R. and D. Gries [1973], *An introduction to programming: A structured approach*, Winthrop, Cambridge, MA.

Dahl, O.J., E.W. Dijkstra, and C.A.R. Hoare [1972], *Structured programming*, Academic Press, New York, NY.

Darlington, J. [July 1975], *Applications of program transformation to program synthesis*, Colloques IRIA on Proving and Improving Programs, Arc-et-Senans, France, pp. 133-144.

Darlington, J. and R.M. Burstall [Mar. 1976], *A system which automatically improves programs*, Acta Informatica, vol. 6, no. 1, pp. 41-60.

Dershowtiz, N. [1978], *The evaluation of programs*, Ph.D. Thesis, Applied Mathematics Dept., Weizmann Institute, Rehovot, Israel.

Dershowitz, N. and Z. Manna [July 1975], *On automating structured programming*, Colloques IRIA on Proving and Improving Programs, Arc-et-Senans, France, pp. 167-193.

Dershowitz, N. and Z. Manna [Nov. 1977], *The evolution of programs: Automatic program modification*, IEEE Software Engineering, vol. SE-3, no. 6, pp. 377-385.

Dershowitz, N. and Z. Manna [May 1978], *Inference rules for program annotation*, Proc. 3rd Intl. Conf. on Software Engineering, Atlanta, GA, pp. 158-167.

Dershowitz, N. and Z. Manna [Aug. 1979], *Proving termination with multiset orderings*, CACM, vol. 22, no. 8, pp. 465-476.

Deutsch, L.P. [May 1973], *An interactive program verifier*, Ph.D. thesis, Univ. of California, Berkeley, CA; Memo CSL-73-1, Xerox Research Center, Palo Alto, CA.

Dijkstra, E.W. [1968], *A constructive approach to the problem of program correctness*, BIT, vol. 8, no. 3, pp. 174-186.

Dijkstra, E.W. [1976], *A discipline of programming*, Prentice Hall, Englewood Cliffs, NJ.

Duran, J.W. [May 1975], *A study of loop invariants and automatic program synthesis*, Ph.D. thesis, Memo SESLTR-12, Software Engineering and Systems Laboratory, Univ. of Texas, Austin, TX.

Elspas, B. [July 1974], *The semiautomatic generation of inductive assertion for proving program correctness*, Interim report, Project 2686, SRI International, Menlo Park, CA.

Eve, J. [Sept. 1975], *On computing the transitive closure of a relation*, Memo CS-75-508, Computer Science Dept., Stanford Univ., Stanford, CA.

Fikes R.E., P.E. Hart, and N.J. Nilsson [Winter 1972], *Learning and executing generalized robot plans*, Artificial Intelligence, vol. 3, no. 4, pp. 251-288.

Floyd, R.W. [1967], *Assigning meanings to programs*, Proc. Symp. in Applied Mathematics, vol. 19 (J.T. Schwartz, ed.), American Mathematical Society, Providence, RI, pp. 19-32.

Floyd, R.W. [Aug. 1971], *Toward interactive design of correct programs*, Proc. Information Processing Cong., Ljubljana, Yugoslavia, pp. 7-10.

Gerhart, S.L. [Apr. 1975], *Knowledge about programs: A model and case study*, Proc. Intl. Conf. on Reliable Software, Los Angeles, CA, pp. 88-95.

Gerhart, S.L. [July 1975], *Verification operator systems and their application to logical analysis of programs*, Colloques IRIA on Proving and Improving Programs, Arc-et-Senans, France, pp. 209-221.

Gerhart, S.L. and L. Yelowitz [Dec. 1976], *Control structure abstractions of the backtracking programming technique*, IEEE Software Engineering, vol. SE-2, no. 4, pp. 285-292.

German, S.M. [May 1974], *A program verifier that generates inductive assertions*, Undergraduate thesis, Memo TR-19-74, Harvard Univ., Cambridge, MA.

German, S.M. [Jan. 1978], *Automating proofs of the absence of common runtime errors*, Conf. Rec. 5th ACM Symp. on Principles of Programming Languages, Tucson, AZ, pp. 105-118.

German S.M., and B. Wegbreit [Mar. 1975], *A synthesizer of inductive assertions*, IEEE Software Engineering, vol. SE-1, no. 1, pp. 68-75.

Gibb, A. [July 1961], *Algorithm 61: Procedures for range arithmetic*, CACM, vol. 4, no. 7, pp. 319-320.

Green, C.C. [Oct. 1976], *The design of the PSI program synthesis system*, Proc. 2nd Intl. Conf. on Software Engineering, San Francisco, CA, pp. 4-18.

Greif, I. and R.J. Waldinger [Apr. 1974], *A more mechanical heuristic approach to program verification*, Proc. Intl. Symp. on Programming, Paris, France, pp. 83-90.

Gries, D. [Nov. 1974], *On structured programming - A reply to Semoliar*, CACM, vol. 17, no. 11, pp. 655-657.

Harrison, W.H. [May 1977], *Compiler analysis of the value ranges for variables*, IEEE Software Engineering, vol. SE-3, no. 3, pp. 243-250.

Hoare, C.A.R. [July 1961], *Algorithm 63: Partition*, CACM, vol. 4, no. 7, p. 321.

Hoare, C.A.R. [Oct. 1969], *An axiomatic basis of computer programming*, CACM, vol. 12, no. 10, pp. 576-580, 583.

Huet, G. and B. Lang [Nov. 1977], *Proving and applying program transformations expressed with second-order patterns*, Report 266, IRIA Laboria, Le Chesnay, France (to appear in Acta Informatica).

Jonassen, A.T. and D.E. Knuth [June 1978], *A trivial algorithm whose analysis isn't*, JCSS, vol. 16, no. 3, pp. 301-322.

Kant, E. [Aug. 1977], *The selection of efficient implementations for a high-level language*, Proc. Symp. on Artificial Intelligence and Programming Languages, Rochester, NY, pp. 140-146.

Katz, S.M. [Sep. 1976], *Invariants and the logical analysis of programs*, Ph.D. thesis, Weizmann Institute of Science, Rehovot, Israel.

Katz, S.M. and Z. Manna [Aug. 1973], *A heuristic approach to program verification*, Adv. Papers 3rd Intl. Conf. on Artificial Intelligence, Stanford, CA, pp. 500-512.

Katz, S.M. and Z. Manna [Apr. 1975], *Towards automatic debugging of programs*, Proc. Intl. Conf. on Reliable Software, Los Angeles, CA, pp. 143-155.

Katz, S.M. and Z. Manna [Dec. 1975], *A closer look at termination*, Acta Informatica, vol. 5, no. 4, pp. 333-352.

Katz, S.M. and Z. Manna [Apr. 1976], *Logical analysis of programs*, CACM, vol. 19, no. 4, pp. 188-206.

King, J.C. [July 1976], *Symbolic execution and program testing*, CACM, vol. 19, no. 7, pp. 385-391.

Kling, R.E. [Aug. 1971], *Reasoning by analogy with applications to heuristic problem solving: A case study*, Ph.D. thesis, Stanford Univ., Stanford, CA.

Knuth, D.E. [Dec. 1974], *Structured programming with go to statements*, Computing Surveys, vol. 6, no. 4, pp. 261-301.

Loveman, D.B. [Jan. 1977], *Program improvement by source-to-source transformation*, JACM, vol. 24, no. 1, pp. 121-145.

Luckham, D.C. and J.R. Buchanan [July 1974], *Automatic generation of programs* *containing conditional statements*, Proc. Conf. on Artificial Intelligence and the Simulation of Behaviour, Sussex, England, pp. 102-126.

Luckham, D.C. and N. Suzuki [Mar. 1977], *Proof of termination within a weak logic of programs*, Acta Informatica, vol. 8, no. 1, pp. 21-36.

McCarthy, J. and C. Talcott [1978], *LISP programming and proving*, Manuscript, Artificial Intelligence Laboratory, Stanford Univ., Stanford, CA.

Manna, Z. [June 1971], *Mathematical theory of partial correctness*, JCSS, vol. 5, no. 3, pp. 239-253.

Manna, Z. and R.J. Waldinger [Summer 1975], *Knowledge and reasoning in program synthesis*, Artificial Intelligence, vol. 6, no. 2, pp. 175-208.

Manna, Z. and R.J. Waldinger [Nov. 1977], *Synthesis: Dreams ⇒ Programs*, Memo AIM-302, Artificial Intelligence Laboratory, Stanford Univ., Stanford, CA.

Manna, Z. and R.J. Waldinger [May 1978], *The logic of computer programming*, IEEE Software Engineering, vol. SE-4, no. 3, pp. 199-229.

Misra, J. [July 1975], *Relations uniformly conserved by a loop*, Colloques IRIA on Proving and Improving Programs, Arc-et-Senans, France, pp. 71-80.

Misra, J. [Sept. 1978], *An approach to formal definitions and proofs of programming principles*, IEEE Software Engineering, vol. SE-4, no. 5, pp. 410-413.

Moriconi, M.S. [Oct. 1974], *Towards the interactive synthesis of assertions*, Memo ATP-20, Automatic Theorem Proving Project, Univ. of Texas, Austin, TX.

Morris, J.H. and B. Wegbreit [Apr. 1977], *Subgoal induction*, CACM, vol. 20, no. 4, pp. 209-222.

Netzer, I. [Apr. 1976], *Logical analysis of recursive programs*, Master's thesis, Weizmann Institute of Science, Rehovot, Israel.

Nelson, C.G. and D. Oppen [Apr. 1978], *Simplification by cooperating decision procedures*, Memo AIM-311, Artificial Intelligence Laboratory, Stanford Univ., Stanford, CA (to appear in CACM).

Sacerdoti, E.D. [Sept. 1975], *The nonlinear nature of plans*, Proc. 4th Intl. Joint Conf. on Artificial Intelligence, Tbilisi, USSR, pp. 206-214.

Sagiv, Y. [Aug. 1976], *A study of the automatic debugging of programs*, Master's thesis, Weizmann Institute of Science, Rehovot, Israel.

Scherlis, W. [May 1974], *On the weak interpretation method for extracting program properties*, Undergraduate thesis, Harvard Univ., Cambridge, MA.

Siklossy, L. [1974], *The synthesis of programs from their properties and the insane heuristic*, Proc. 3rd Texas Conf. on Computing Systems, Austin, TX.

Sintzoff, M. [Jan. 1972], *Calculating properties of programs by valuations on specific models*, Proc. ACM Conf. on Proving Assertions About Programs, Las Cruces, NM, SIGPLAN Notices, vol. 7, no. 1, pp. 203-207.

REFERENCES

Standish, T.A., D.C. Harriman, D.F. Kibler, and J.M. Neighbors [Feb. 1976], *Improving and refining programs by program manipulation*, Memo, Dept. of Information and Computer Science, Univ. of California, Irvine, CA.

Sussman, G.J. [1975], *A computer model of skill acquisition*, American Elsevier, New York, NY.

Suzuki N. and K. Ishihata [Jan. 1977], *Implementation of an array bound checker*, Conf. Rec. 4th ACM Symp. on Principles of Programming Languages, Los Angeles, CA, pp. 132-143.

Tamir, M. [Aug. 1976], *ADI - Automatic derivation of invariants*, Master's thesis, Weizmann Institute of Science, Rehovot, Israel.

Teitelman, W. [1974], *INTERLISP reference manual*, Xerox Research Center, Palo Alto, CA.

Ulrich, J.W. and R. Moll [Aug. 1977], *Program synthesis by analogy*, Proc. ACM Symp. on Artificial Intelligence and Programming Languages, SIGPLAN Notices, vol. 12, no. 8, Rochester, NY, pp. 22-28.

Waldinger, R.J. [1977], *Achieving several goals simultaneously*, in Machine Intelligence 8: Machine Representations of Knowledge (E.W. Elcock and D. Michie, eds.), Ellis Horwood, Chichester, England, pp. 94-136.

Waldinger, R.J. and K.N. Levitt [Fall 1974], *Reasoning about programs*, Artificial Intelligence, vol. 5, no. 3, pp. 235-316.

Warren, D.H.D. [July 1976], *Generating conditional plans and programs*, Proc. Conf. on Artificial Intelligence and Simulation on Behaviour, Edinburgh, Scotland, pp. 344-354.

Waters, R.C. [July 1977], *A method, based in plans, for understanding how a loop implements a computation*, Working paper 150, Artificial Intelligence Laboratory, MIT, Cambridge, MA.

Wegbreit, B. [Feb. 1974], *The synthesis of loop predicates*, CACM, vol. 17, no. 2, pp. 102-112.

Wegbreit, B. [Sept. 1975], *Property extraction in well-founded property sets*, IEEE Software Engineering, vol. SE-1, no. 3, pp. 270-285.

Wegbreit, B. [Jan. 1976], *Goal-directed program transformation*, Conf. Rec. 3rd ACM Symp. on Principles of Programming Languages, Atlanta, GA, pp. 153-170.

Wegbreit, B. and J.M. Spitzen [Apr. 1976], *Proving properties of complex data structures*, JACM, vol. 23, no. 2, pp. 389-396.

Wensley, J.H. [Jan. 1959], *A class of non-analytical iterative processes*, Computer J., vol. 1, no. 4, pp. 163-167.

Wilber, B.M. [Mar. 1976], *A QLISP reference manual*, Technical note 118, Artificial Intelligence Center, SRI International, Menlo Park, CA.

Wirth, N. [1973], *Systematic programming: An introduction*, Prentice-Hall, Englewood Cliffs, NJ.

Wirth, N. [Dec. 1974], *On the composition of well-structured programs*, Computing Surveys, vol. 6, no. 4, pp. 247-259.

Yelowitz, L. and A.G. Duncan [Aug. 1977], *Abstractions, instantiations and proofs of marking algorithms*, Proc. Conf. on Artificial Intelligence and Programming Languages, SIGPLAN Notices, vol. 12, no. 8, Rochester, NY, pp. 13-21.

LIST OF PUBLICATIONS

(partially supported by grant AFOSR-78-3483)

Journals and Conference papers:

Dershowitz, N. and Z. Manna [Nov. 1977], *The evolution of programs: A system for automatic program modification*, IEEE Transactions on Software Engineering, Vol. SE-3, No. 6. Also presented at the ACM symposium on principles of Programming Languages, Los Angeles, Jan. 1977.

Dershowitz, N. and Z. Manna [May 1978], *Inference rules for program annotation*, Proceedings of the Third International Conference on Software Engineering, Atlanta, GA.

Dershowitz, N. and Z. Manna [Aug. 1979], *Proving termination with multiset orderings*, CACM, Vol. 22, No. 8.

Theses:

Netzer, I. [April 1976], *Logical analysis of recursive programs*, Master's thesis, Weizmann Institute.

Tamir, M. [Aug. 1976], *ADI: automatic derivation of invariants*, Master's thesis, Weizmann Institute. Also presented at the Computer Science Conference, Atlanta, GA, Jan 1977.

Sagiv, Y. [Aug. 1976], *A study of the automatic debugging of programs*, Master's thesis, Weizmann Institute.

Fried, R. [Feb. 1977], *A multi-processing control structure to facilitate search methods for QLISP goal-trees*, Master's thesis, Weizmann Institute.

Weingarten, Y. [May 1978], *Intermittent assertion proof schemes*, Master's thesis, Weizmann Institute.

Dershowitz, N. [Nov. 1978], *The evolution of programs*, Ph.D. thesis, Weizmann Institute.

Technical Reports:

Raim, M. [Oct. 1976], *The QLISP/370 reference manual*, Technical report, Weizmann Institute.

Raim, M. [May 1977], *A guide to INTERLISP/370*, Technical report, Weizmann Institute.

Raim, M. [July 1977], *The QLISP transplant operation*, Technical report, Weizmann Institute.